155-158-10
OT-851-55

Centro de Investigación y de Estudios Avanzados
del Instituto Politécnico Nacional
Unidad Guadalajara

Implementación Física de la Red Neuronal NARX para Aplicaciones en Predistorsión Digital

Tesis que presenta:
Juan Antonio Renteria Cedano

para obtener el grado de:
Maestro en Ciencias

en la especialidad de:
Ingeniería Eléctrica

Director de Tesis
Dra. Susana Ortega Cisneros

CINVESTAV del IPN Unidad Guadalajara, Guadalajara, Jalisco, Agosto de 2014.

Implementación Física de la Red Neuronal NARX para Aplicaciones en Predistorsión Digital

Tesis de Maestría en Ciencias
Ingeniería Eléctrica

Por:

Juan Antonio Renteria Cedano

Ingeniero en Comunicaciones y Electrónica
Universidad de Guadalajara 2012-2014

Becario de CONACYT, expediente no. 486182/280973

Director de Tesis

Dra. Susana Ortega Cisneros

CINVESTAV del IPN Unidad Guadalajara, Agosto de 2014.

DEDICATORIA

El presente trabajo de tesis lo dedico a mi madre, a mi padre y mis hermanos por su apoyo y cariño incondicional, por los consejos y las enseñanzas que me hicieron crecer como persona de bien para la sociedad, agradezco las palabras de aliento que me brindaron para seguir adelante en los momentos difíciles, por enseñarme a no rendirme y siempre luchar para lograr mis metas.

AGRADECIMIENTOS

Agradezco a mi mamá: María de los Ángeles Cedano, a mi papá: Juan Rentería, mi hermana: Yesenia Aideé, mi hermano: José Efraín y a mi cuñado: Arturo García por brindarme el apoyo incondicional en todo momento y por estar siempre pendientes en los avances de mis estudios de postgrado.

Agradezco a mis viejos amigos por su apoyo y los nuevos amigos agradezco su apoyo, consejos y paciencia.

Agradezco a los Doctores de Diseño Electrónico por los consejos y enseñanzas durante mi formación y a mi directora de tesis la Dra. Susana Ortega Cisneros por las enseñanzas, consejos y facilidades con la herramienta de trabajo.

Al Consejo Nacional de Ciencia y Tecnología CONACyT por las facilidades brindadas para continuar y finalizar mis estudios por medio de la beca nacional para posgrado.

Finálmente agradezco al Centro de Investigación de Estudios Avanzados del IPN CINVESTAV Unidad Guadalajara por darme la oportunidad de ingresar a la institución y continuar con mi preparación profesional.

RESUMEN

En el presente trabajo de tesis se muestra la implementación de un predistorsionador digital basada en una red neuronal NARX configurada en un dispositivo FPGA de Xilinx con aplicación en linealización de amplificadores de potencia de RF. Los amplificadores de potencia por naturaleza son dispositivos no lineales y el empleo de un método de linealización, impactaría en el desarrollo de dispositivos más eficientes y con menor consumo de energía.

El enfoque de las redes neuronales para predistorsión digital en FPGA, es aprovechar la característica que permite modelar sistemas no lineales y los efectos de memoria en un dispositivo portátil.

El modo de funcionamiento de la arquitectura es *offline*, por lo que el predistorsionador no funciona en lazo cerrado, las mediciones del amplificador son almacenadas en el equipo de cómputo y el procesamiento matemático es realizado por el FPGA.

Para realizar una arquitectura con un procesamiento matemático complejo, es necesario fraccionar el diseño en bloques de menor complejidad, analizar la metodología de implementación de cada uno de ellos y seleccionar el formato numérico que presente una mayor precisión de cálculo, característica que permitirá reducir considerablemente el error de los resultados, esta metodología fue aplicada para el desarrollo del predistorsionador digital en FPGA.

El desarrollo del sistema fue implementado por la herramienta "*ISE Design Suite*" de Xilinx empleando la tarjeta de evaluación ML605 que contiene un FPGA Virtex-6 de gran capacidad en recursos lógicos.

Se realizó una implementación de la red neuronal NARX en Matlab lo que permitió tener un punto de comparación de los resultados obtenidos con respecto al FPGA, y por medio de un software de simulación de RF

"*SystemVue*", se pueden obtener las características del modelo inverso de los resultados por ambos métodos.

Y por último se realiza un análisis de correspondencia de los resultados obtenidos por ambos métodos implementados.

ABSTRACT

In this thesis, we show the implementation of a digital predistorter based on a NARX neural network with an RF linear power amplifier. The power amplifiers are non-linear devices by nature and the use of a linearization method would impact the development of power efficient devices.

The focus of neural networks for digital predistortion on FPGAs is to take advantage of the characteristic that allows modeling non-linear systems and the effects of memory on a portable device.

The architecture's operating mode is offline, so the predistorter does not function in closed-loop. The measurements of the amplifier are stored in the computing device and the mathematical processing is realized by the FPGA.

In order to build an architecture with a complex mathematical process, it is necessary to split the design into less complex blocks, analyze the implementation methodology of each, and select the numerical format that presents the most precise calculation. This methodology was applied for the development of the digital predistorter on an FPGA.

The development of this system was implemented on the Xilinx "ISE Design Suite" tool using the ML605 evaluation kit, which contains a Virtex-6 FPGA with high capacity in logical resources.

The implementation of the NARX neural network was realized in Matlab, which permits a comparison point of the results obtained with respect to the FPGA, using an RF "SystemVue" software simulation. The inverse model main characteristics can be obtain by means of the results of both methods.

Finally, a matching analysis was performed between results obtained using both implemented methods.

CONTENIDO

1. H. T. Siefelmann, B. G. Horne and C. L. Giles, "Computational capabilities of recurrent NARX neural networks," *IEEE Transactions on Systems, Man and Cybernetics-Part B: Cybernetics,* Vol. 27, No.2, pp. 208-215, Apr. 1997.....93

2. L.M. Aguilar-Lobo, A. Garcia-Osoria, J.R. Loo-Yau, S. Ortega-Cisneros, P. Moreno, J.E. Rayas-Sanchez, J.A. Reynoso-Hernández, "A Digital Predistortion Technique Based on a NARX Network to Linearize GaN Class F Power Amplifiers", *IEEE* 57th International Midwest Symposium on Circuits And Systems, August 2014.
93

3. M. Bahoura, C.-W. Park, "FPGA-Implementation of an Adaptive Neural Network for RF Power Amplifier Modeling", in *New Circuits and Systems Conference (Newcas). 2011 IEEE 9th International, june 2011, pp.29-32.*.....93

4. M. Atencia, H. Boumeridja, G. Joya, F. Garcia-Lagos and F.Sandoval, "FPGA Implementation of a Systems Identification Module Based Upon Hopfield Networks", *Neurocomputing,* 70(2007) 2828-2835, 2007.....93

5. J.L. Bastos, H.P. Figueroa, A. Monti, "FPGA Implementation of Neural Networks-Based Controllers for Power Electronics Applications", *in Applied Power Electronics Conference and Exposition, 2006. APEC '06. Twenty-First Annual IEEE,* 2006, pp. 1–6.....93

6. A.L.S. Braga, C.H. Llanos, D. Gohringer, J. Obie, J. Becker, M. Hubner, "Performance, Accuracy, Power Consumption and Resource Utilization Analysis for Hardware/Software realized Artificial Neural Networks", *Bio-inspired Computing: Theories and Aplications (BIC-TA), 2010 IEEE Fifth Internatiol Conference on,* pp.1629-1636. Sept.2010. 93

7. "Virtex-6 FPGA ML605 Evaluation Kit", www.xilinx.com/products/boards-and-kits/EK-V6-ML-605-G.htm, Xilinx Inc.....93

8. K. Mohamad, M. F. O. Mahmud, F. H. Adnan, W. F. H. Abdullah. "Design of single neuron on FPGA", Humanities, Science and Engineering Research (SHUSER) 2012 IEEE Symposium on.....93

9. IEEE Standard for Floating-Point Arithmetic (IEEE Std 754-2008), Revision of IEEE Std 754-4985. 29 August 2008.....93

10. Meng Qian, "Application of CORDIC Algorithm to Neural Networks VLSI Design", *in Multconf. Computational Engineering in Systems Applications IMACS.* Oct. 2006, pp.504-508. 93

11. Z.Salcic, A. Smailagic, "Digital system design and prototyping using field programmable logic", Boston: Kluwer Academic Publishers, 1997, pp. 134-141. 93

12. M. A. Sartin, A. C. R. da Silva "Approximation of Hyperbolic Tangent Activation Function Using Hybrid Methods", Department of Computing, UNEMAT-Universidade do Estado de Mato Grosso, Colider, MT, Brazil. 93

14. *ICANN 2005*, LNCS 3697, pp. 559-564, 2005.G. B. Wakhle, I. Aggarwal, S. Gaba, "Synthesis and Implementation of UART Using VHDL Codes", *International Symposium on Computer, Consumer and Control*, 2012.....93

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

Capítulo 1

Introducción

Los amplificadores de potencias son dispositivos esenciales en los sistemas de comunicación y en aplicaciones de VHF donde se requieren elevadas potencias de RF, motivo por el cual se pretenden dispositivos con altos rendimientos y salidas sin distorsión. Para conseguir estas características en los amplificadores de RF como los de clase D, E y F se emplean los transistores como conmutadores debido a que por naturaleza son inherentemente no lineales. Para conseguir la linealidad de los dispositivos, se han desarrollado métodos que corrigen esta característica como son: *feedfordward*, *feedback* y predistorsión.

1.1 Introducción a predistorsión digital

EL realizar una corrección de la respuesta no lineal de los amplificadores de potencia es un problema que va con el creciente uso de avanzadas técnicas de modulaciones de ondas espectrales empleadas en los sistemas de comunicación. Un amplificador de potencia que trabaja con señales de banda ancha provoca emisiones fuera de la banda de interés, también conocida como recrecimiento espectral, provocado por la no linealidad del amplificador. Típicamente el amplificador se hace funcionar lo más cerca posible de la región de saturación para lograr la máxima potencia y eficiencia, lo que significativamente funciona en su región no lineal. Lo que implica una distorsión significativa y un recrecimiento espectral. Una solución para el dilema de eficiencia-linealidad, son las técnicas que han sido propuestas en la literatura para linealizar los amplificadores de RF. Algunas de estas

son: *feedfordward, feedback,* predistorsión analógica y sus variantes optimizadas que son basadas en corrección analógica donde la aplicación de esta técnica en el transmisor es incomoda, debido a la complejidad de los circuitos, problemas de estabilidad e insuficiencia en la linealización.

La técnica de predistorsión digital es ahora ampliamente aceptada ya que proporciona alta precisión en la síntesis de la función de predistorsión y presenta capacidad de reconfiguración para hacerlo adecuado en la solución de diversos problemas. El predistorsionador digital (DPD) es un bloque que se antepone a la entrada de un amplificador de potencia y la principal característica es contener la función inversa de la respuesta entrada-salida del amplificador de potencia con la finalidad de corregir la salida no lineal. Para que un DPD sea genérico a cualquier tipo de amplificador, se realizan diseños con algoritmos adaptativos, para ello el uso de dispositivos reconfigurables especializados en el procesamiento digital de señales ha cobrado gran importancia en el tema. A lo largo del tiempo se han desarrollado diferentes técnicas de predistorsión digital como son: series de Volterra[1], memorias polinomiales[2], y redes neuronales artificiales (ANNs).

Las redes neuronales artificiales son sistemas basados en una representación simplificada de estructura y funcionamiento del sistema nervioso, ya sea simulado en software o construido en hardware. Se deben entrenar mediante ejemplos conocidos hasta que son capaces de asociar patrones de entrada con respuestas definidas sin necesidad de una programación explicita para un patrón en particular. Esta característica permite a las redes resolver problemas nuevos donde incluso se crean nuevas categorías de patrones para los cuales no es posible diseñar algoritmos de solución, con lo cual pueden enfrentar con éxito casos en que no es posible definir una relación entre las causas y los efectos. Las redes neuronales son modelos de cómputo paralelo que se conocen en ingeniería con el nombre de redes neuronales artificiales. Ofrecen soluciones a problemas donde es difícil aplicar lo métodos tradicionales.

Típicamente el uso de redes neuronales era llevado por un equipo de cómputo y software especializado para el procesamiento matemático, pero debido a la compleja portabilidad de éste, se han buscado alternativas de implementación de estos sistemas. Existen componentes que presentan las características de ser reconfigurables, principalmente las compuertas en arreglo programables en campo (FPGA: *Field Programmable Gate Array*), las cuales son dispositivos electrónicos digitales de aplicación general, comúnmente consisten en una matriz bidimensional de bloques lógicos configurables que se pueden conectar mediante recursos de interconexión también programables. Estos

---

[1] Fehri, Bilel, and Slim Boumaiza. "Baseband equivalent Volterra series for digital predistortion of dual-band power amplifiers." (2014): 1-15.

2 Morgan, Dennis R., Zhengxiang Ma, Jaehyeong Kim, Michael G. Zierdt, and John Pastalan. "A generalized memory polynomial model for digital predistortion of RF power amplifiers." Signal Processing, IEEE Transactions on 54, no. 10 (2006): 3852-3860.

recursos incluyen segmentos de pista de diferentes longitudes e interruptores con los que se pueden enlazar bloques con pistas, segmentes de pistas con otras pistas y bloques de entrada/salida (E/S) con pistas. En estos dispositivos se pueden implementar desde funciones básicas sencillas como una compuerta lógica hasta sistemas en chip (SoC: *System on Chip*).

Aprovechando la capacidad de reconfiguración del FPGA, se han implementado redes neuronales capaces de resolver diversos problemas, como son: modelado de sistemas no lineales, identificación de sistemas dinámicos, reconocimiento de patrones, predicción de eventos, procesamiento de imágenes, entre otros.

En este trabajo de tesis, se presenta la implementación de un predistorsionador digital para un amplificador de potencia de RF empleando un señal LTE centrada a una frecuencia de 2 GHz, aplicando una red neuronal NARX en un dispositivo reconfigurable FPGA de Xilinx.

1.2 Justificación

Este trabajo de tesis pretende servir como referencia para trabajos futuros de investigación en aplicación de redes neuronales en dispositivos reconfigurables aplicados a sistemas de predistorsión digital de amplificadores de potencia de RF. Se presenta una metodología de la implementación de la arquitectura de la red neuronal NARX, un nuevo método de implementación de la función de activación y una comparación de los resultados obtenidos por medio del FPGA contra los procesados en Matlab.

1.3 Planteamiento del problema

Los amplificadores de potencia de RF tiene la naturaleza de ser no lineales, y para lograr la máxima potencia y eficiencia de estos se hacen trabajar lo más cerca de la región lineal, para el caso de los amplificadores de banda ancha esto provoca emisiones fuera de la banda generando una distorsión conocida como recrecimiento espectral[3], por lo que es necesario aplicar una técnica de corrección que permita mejorar el rendimiento y disminuir la distorsión en la salida del amplificador. Un medio de solución a este problema es la aplicación de redes neuronales. Usualmente este tipo de algoritmo matemático es realizado por equipo de cómputo por lo que carece el sistema de portabilidad, debido a ellos se han buscado de alternativas que sean capaces de resolver este problema

[3] Rawat, Meenakshi, Karun Rawat, and Fadhel M. Ghannouchi. "Adaptive digital predistortion of wireless power amplifiers/transmitters using dynamic real-valued focused time-delay line neural networks." Microwave Theory and Techniques, IEEE Transactions on 58, no. 1 (2010): 95-104.

donde el uso de dispositivos reconfigurables han cobrado importancia, ya que pueden ser configurados para reproducir operaciones matemáticas complejas.

El trabajo de tesis que se presenta es el desarrollo de una red neuronal con arquitectura NARX para la corrección de la linealidad de un amplificador de potencia de RF que emplea una señal de banda ancha LTE centrada a 2 GHz[4].

1.4 Hipótesis

Es posible implementar un predistorsionador digital empleando una red neuronal NARX en FPGA, usando lenguaje Verilog para generar bloques a medida reduciendo el consumo de recursos en comparación de sistemas implementados en procesadoreś. Se aprovecha del formato coma flotante para generar un punto de comparación con el procesamiento matemático realizado en Matlab.

1.5 Objetivos

1.5.1 Objetivo general

Diseñar, implementar y aplicar la técnica de predistorsión digital empleando redes neuronales en dispositivos reconfigurables FPGA, el diseño debe de ser capaz de procesar los datos obtenidos de un amplificador de potencia de RF y calcular la función inversa de la relación entrada-salida.

1.5.2 Objetivos específicos

- Realizar revisión del estado del arte en relación a la implementación de arquitecturas empleadas en predistorsión digital y aplicación de redes neuronales en FPGA.
- Documentar y seleccionar el formato numérico a emplear: punto fijo o coma flotante.
- Revisión de implementaciones de función de activación.
- Implementación de una neurona simple.
- Implementación de la arquitectura de la red neuronal NARX.
- Realizar simulaciones, pruebas y comparación de resultados de la red neuronal en FPGA y Matlab.

---

[4] L.M. Aguilar-Lobo, J.R. Loo-Yau, S. Ortega-Cisneros, "A Digital Predistortion Technique Based on a NARX Network to Linearize GaN Class F Power Amplifiers", *IEEE* 57th International Midwest Symposium on Circuits And Systems, August 2014.

1.6 Metodología

La metodología para el desarrollo de esta tesis se muestra a continuación.

En primer término realizar revisión del estado del arte para obtener los fundamentos del funcionamiento y nomenclatura de redes neuronales y predistorsión digital. Analizar investigaciones realizadas sobre aplicaciones de las redes neuronales en dispositivos reconfigurables.

Analizar la complejidad de implementación de las operaciones requeridas para la red neuronal y decidir el tipo de formato numérico a emplear.

Analizar los métodos empleados para el cálculo de las funciones de activación e implementar el que ofrezca mejor característica de precisión y desempeño.

Implementar una neurona simple y posteriormente alambrar la red neuronal requerida para el proceso de la información del amplificador de potencia.

Realizar la interfaz de comunicación entre el FPGA y el equipo de cómputo para la transmisión de información, posteriormente comparar resultados obtenidos con el procesamiento en hardware y software.

Realizar simulación de resultados y graficar las curvas características de salida del predistorsionador empleando software de análisis de RF.

El desarrollo de la tesis se ha divido en un conjunto de siete capítulos, de se presenta el marco teórica y el flujo de diseño empleado.

A continuación se muestra una descripción del contenido de los capítulos de este trabajo de tesis.

En el primer capítulo se describen los alcances del proyecto, la justificación de su implementación, los objetivos y la metodología de diseño de la arquitectura.

En el segundo capítulo se presentan los antecedentes y arquitecturas que se han desarrollado en este campo de estudio para la corrección de la no linealidad de los amplificadores de potencia.

En el tercer capítulo se realiza una descripción de las redes neuronales y los bloques básicos que la componen, algunas arquitecturas relevantes y la nomenclatura estándar empleada.

En el cuarto capítulo se presenta el kit de evaluación utilizado para el desarrollo del sistema, se muestra la arquitectura de la red neuronal empleada y los algoritmos de los bloques básicos, se presenta un nuevo método de implementación de una tangente hiperbólica en FPGA empleada como función de activación. Por último se presenta el diseño de la red neuronal y su unidad de control.

En el quinto capítulo se muestra la arquitectura final del predistorsionador digital, así como una descripción de los bloques que lo constituyen, se proporciona los pasos de programación de una red neuronal en Matlab de esta manera se tiene medio de comparación de los resultados obtenidos por ambos métodos.

En el sexto capítulo se presentan resultados obtenidos del sistema en el FPGA y en Matlab, analizando la correspondencia de los resultados procesándolos con un software de simulación de RF.

Y por último, en el séptimo capítulo se presentan las conclusiones, el trabajo a futuro que permiten mejorar el diseño y las publicaciones realizadas del trabajo desarrollado.

Capítulo 2

Marco teórico

El amplificador de potencia es un bloque que tiene mayor importancia en sistemas de comunicación y dispositivos móviles, pero a su vez este circuito consume mayor energía que cualquier otro componente. Las modernas formas de modulaciones espectrales, requieren de componentes que presente una mayor eficiencia y linealidad, pero debido a su naturaleza los amplificadores de potencia son no lineales, por lo que es necesario implementar alguna técnica de linealización. En este capítulo se describen algunos antecedentes de arquitecturas que se han aplicado en linealización de amplificadores de potencia.

2.1 Introducción

EL amplificador de potencia es el circuito que consume más energía que cualquier otro en un dispositivo electrónico móvil y estos representan un factor importante en el gasto de operación para el proveedor de servicios.

Las modernas formas de modulaciones espectrales requieren de la máxima linealidad de los amplificadores de potencia y para ello debe de ser trabajado por debajo de la saturación, para mejorar la eficiencia de los amplificadores, los diseñadores de circuitos electrónicos utilizan técnicas digitales para reducir el factor de cresta y mejorar la linealidad de los amplificadores, lo que permite que se ejecute más cerca de la saturación; por lo que la técnica de predistorsión digital se ha convertido en método revolucionario para la linealización de los amplificadores.

2.2 Técnicas de linealización

Las principales técnicas de linealización de amplificadores de potencia se pueden dividir en dos grupos, que dependen de la metodología que emplean para la corrección de la señal.

Métodos para evitar la distorsión: En método LINC (*Linear amplification with nonlinear components*) la señal original con envolvente variante en el tiempo es transformada en dos señales de envolventes constantes. Estas señales son amplificadas por separado y sin distorsión y posteriormente son recombinadas produciente una réplica amplificada de la señal original.

El método *EE&R* (*Envelope Elimination and Restoration*) separa la señal con envolvente variable en sus componentes polares de amplitud y fase. La amplitud es utilizada para modular la tensión de alimentación del amplificador y la fase es incorporada por una señal con envolvente constante que es amplificada sin distorsión en el amplificador.

Métodos para reducir la distorsión: Aquí se incluyen los sistemas donde se compensa la distorsión introducida por el amplificador. Esta compensación se puede efectuar tomado una muestra de los productos de intermodulación de la señal obtenida del amplificador e inyectándola apropiadamente en la salida esta técnica es conocida como *feedfordward*, otra técnica es tomar una muestra de la señal de salida e introducirla a la entrada llamada *feedback* y otro método que ha cobrado gran importancia es realizar una alteración apropiada de la forma de la envolverte de la señal de entrada llamado *predistorsión*.

2.3 Métodos para evitar la distorsión

2.3.1 Amplificación lineal con componentes no lineales (*LINC*)

La técnica LINC (*Linear amplification with Nonlinear Components*) toma ventaja de la alta eficiencia de los amplificadores de potencia conmutados, y consiste en descomponer la señal variante en amplitud, en señales de envolvente constante que son amplificadas empleando dispositivos altamente no lineales sin introducir distorsión.

La modulación de amplitud es recuperada a la salida de los amplificadores de potencia, los cuales teóricamente garantizan alta eficiencia y buena linealidad.

La combinación de potencias de salida de los amplificadores es el factor clave para determinar la linealidad y eficiencia en el sistema. La técnica LINC descompone las amplitudes variantes de las señales en dos señales de envolvente constante desfasado. Después de amplificar las señales, son

combinadas una con la otra para generar una versión lineal amplificada de la señal de entrada[5].

Dependiendo de la topología del combinador de señales, ya sea la eficiencia o la linealidad es significativamente degradada.

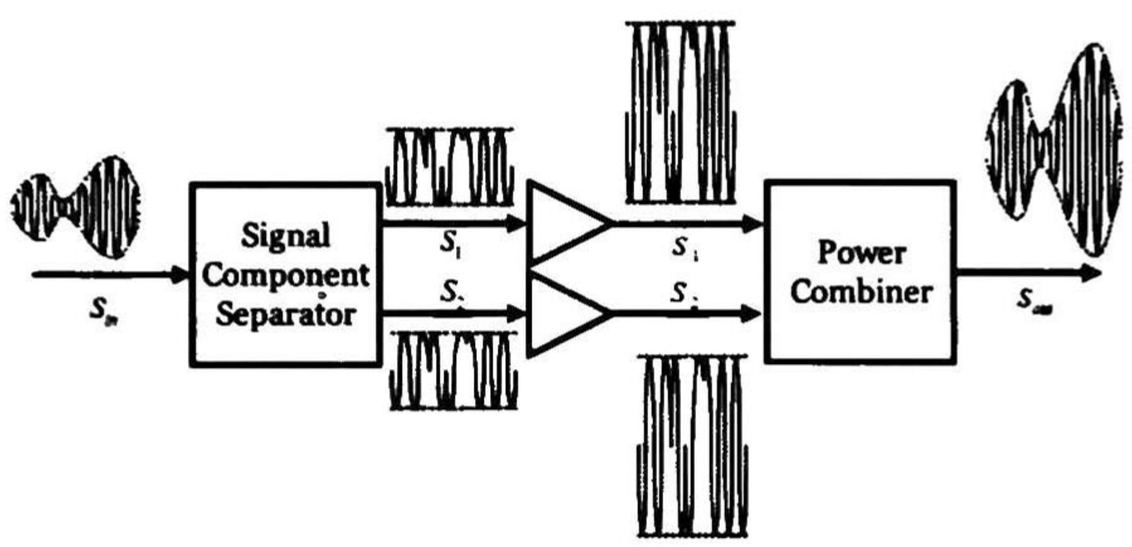


Fig. 2.1 Diagrama a bloques simplificado de un transmisor LINC.

2.3.2 Eliminación de envolvente y restauración (EE&R)

La técnica EE&R (*Envelope Elimination and Restoration*), como su nombre lo indica consiste en que la envolvente de entrada de la señal de RF es eliminada por un limitador para generar una señal de fase de amplitud constante. Al mismo tiempo, la información de la magnitud es extraída por un detector de envolvente. La información de la magnitud y fase son amplificadas por separado y estas son recombinadas para restaurar la señal de RF deseada. Una manera de recombinar los componentes de la magnitud y fase de la señal es empleando un eficiente amplificador de RF conmutado. Las componentes de magnitud y fase pueden ser recombinadas si la señal de fase de RF es aplicada en la compuerta del transistor y la magnitud de la señal (señal de baja frecuencia) modula directamente a la fuente[6]

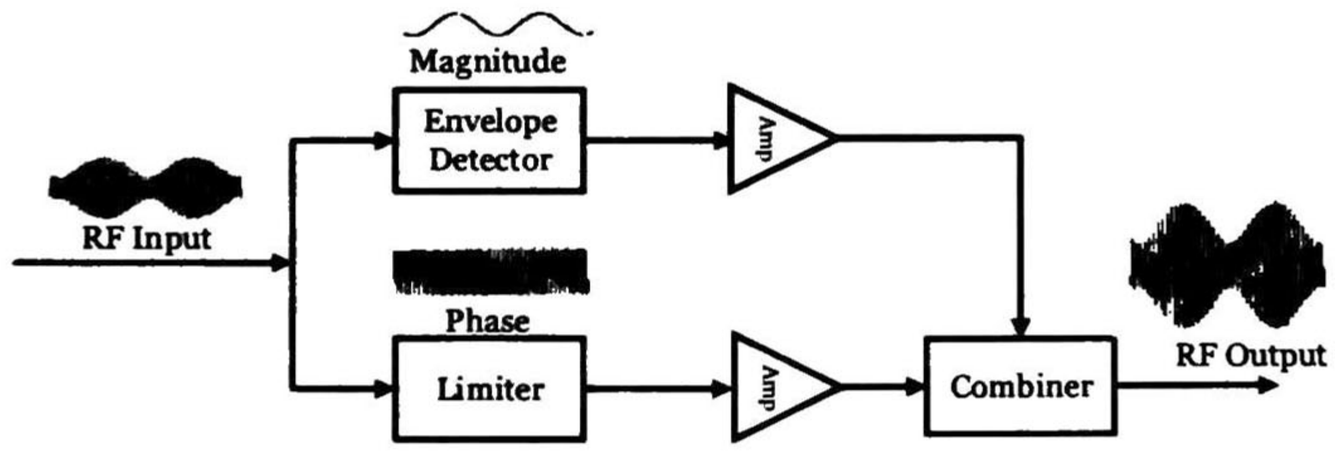


Fig. 2.2 Diagrama a bloques de un sistema EE&R.

---

[5] Helaoui, Mohamed, and Fadhel M. Ghannouchi. "Linearization of power amplifiers using the reverse mm-linc technique." Circuits and Systems II: Express Briefs, IEEE Transactions on 57, no. 1 (2010): 6-10.

[6] Su, David K., and William J. McFarland. "An IC for linearizing RF power amplifiers using envelope elimination and restoration." Solid-State Circuits, IEEE Journal of 33, no. 12 (1998): 2252-2258

2.4 Métodos para reducir la distorsión

2.4.1 Feedback

La realimentación (*feedback*) negativa es ampliamente utilizada en el diseño de amplificadores ya que presenta beneficios como la estabilización de la ganancia del amplificador frente a variaciones de los dispositivos, temperatura, variaciones en fuente de alimentación y degradación de los componentes. Permite al diseñador ajustar la impedancia de entrada y salida del circuito sin tener que realizar demasiadas modificaciones. La disminución de la distorsión y el aumento del ancho de banda hacen que la realimentación negativa sea una opción en linealización de amplificadores. Algunos de los inconvenientes que presenta esta técnica de linealización es que la ganancia del amplificador disminuye en la misma proporción al aumento de la linealidad y ancho de banda, este problema se resuelve incrementado las etapas amplificadoras, por otro lado al implementar la realimentación de un circuito genera la tendencia a la oscilación[7].

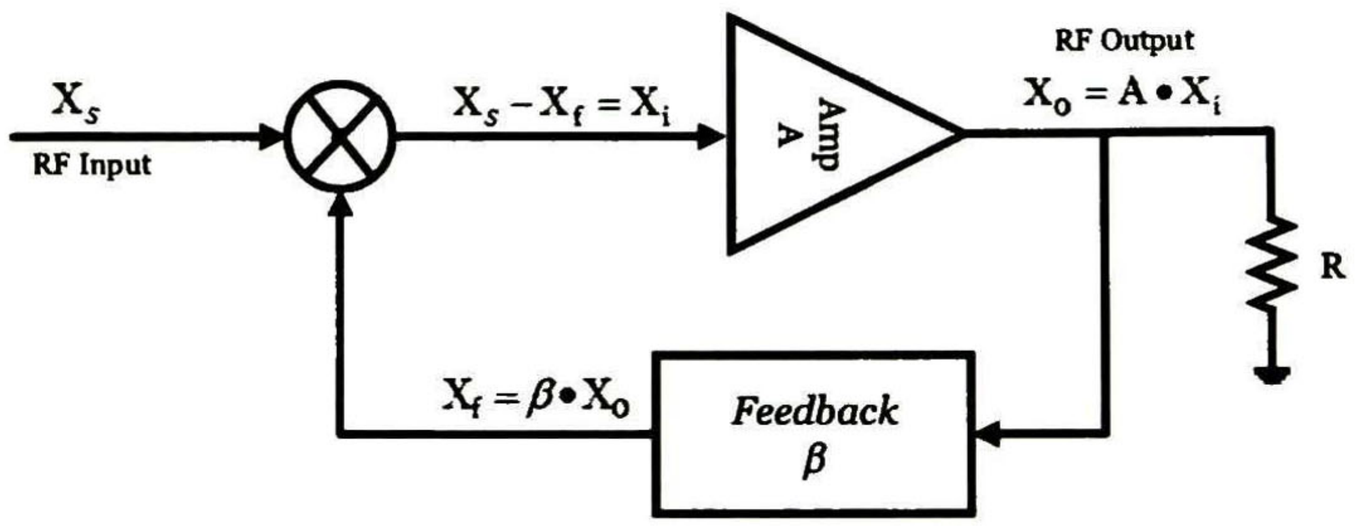


Fig. 2.3 Diagrama a bloques de un circuito *feedback*.

2.4.2 Feedforward

Es la técnica de linealización más eficaz empleada hoy en día en los sistemas de comunicaciones multiportadoras de frecuencias elevadas[8]. Ofrece excelentes prestaciones de ancho de banda y de reducción de distorsión. La arquitectura de un linealizador *feedforward* consta de dos circuitos fundamentales: el circuito de cancelación de la señal y el circuito de cancelación del error. En el primero de estos circuitos se obtiene una señal de error que contiene los productos de intermodulación que produce el amplificador de potencia. Esta señal se obtiene de la

---

[7] Shi, Bo, and Lars Sundstrom. "Linearization of RF power amplifiers using power feedback." In Vehicular Technology Conference, 1999 IEEE 49th, vol. 2, pp. 1520-1524. IEEE, 1999.

[8] Cripps, Steve C. Advanced techniques in RF power amplifier design. Artech House, 2002.

comparación con un combinador donde se toma una muestra de la señal de salida propiamente atenuada y la señal de entrada retardada.

En el circuito de cancelación error[9], la señal obtenida es amplificada apropiadamente en el amplificador del error y es inyectada en contrafase a la salida para cancelar los productos de intermodulación que presenten en la salida del amplificador. Antes de combinar la señal de salida del amplificador es conveniente retardarla. La combinación de la señal de error con la señal de salida del amplificador de potencia suele hacerse en un acoplador direccional de potencia. El amplificador del error debe operar en un modo suficientemente lineal para no añadir distorsión al sistema. Los aspectos claves de esta técnica son los desequilibrios de amplitud y fase, así como la desigualdad de los retardos de las señales al pasar entre las diferentes ramas que se comparan. En aplicaciones de RF estos desajustes pueden comprometer las prestaciones de linealización, por lo que se hace necesario incluir algún circuito de compensación[10] para mantener las prestaciones ofrecidas por esta técnica de linealización.

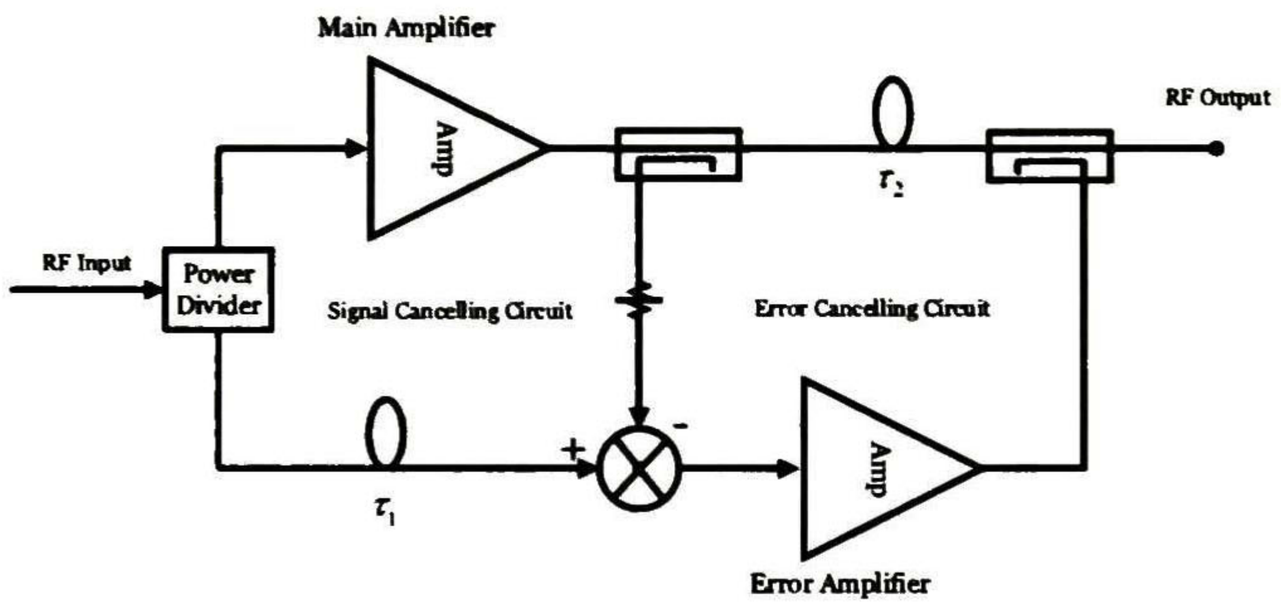


Fig. 2.4 Diagrama de un linealizador *feedforward*.

2.4.3 Predistorsión

Esta técnica trata de modificar las características de la señal modulada antes de pasar por la parte no lineal del sistema hacia el amplificador de potencia. La señal de entrada es modificada mediante una regla de predistorsión, donde el amplificador de potencia se hace funcionar a través de un dispositivo no lineal cuya relación entrada-salida es la inversa de la relación entrada-salida del amplificador.

---

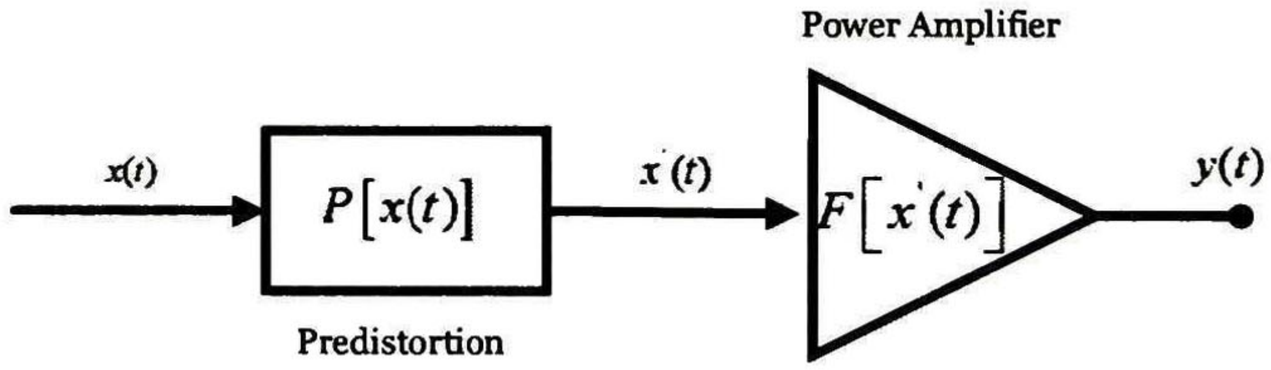
[9] Honarvar, M. A., M. N. Moghaddasi, and A. R. Eskandari. "Power amplifier linearization using feedforward technique for wide band communication system." In Radio-Frequency Integration Technology, 2009. RFIT 2009. IEEE International Symposium on, pp. 72-75. IEEE, 2009.

[10] Abuelma'atti, Muhammad Taher, Abdullah MT Abuelmaatti, Tk Yeung, and G. Parkinson. "Linearization of GaN power amplifier using feedforward and predistortion techniques." In Quality Electronic Design (ASQED), 2011 3rd Asia Symposium on, pp. 282-287. IEEE, 2011.

Por lo general, se trata de un algoritmo que actúa sobre la señal que se pretende transmitir en base a coeficientes de un modelo estime las no linealidades del amplificador. En un predistorsionador son de suma importancia el modelado de la parte no lineal del sistema y el método de ajuste de los coeficientes que sea empleado para conseguir un resultado lineal.

Según la etapa de frecuencia en que la función del predistorsión es implementada, tenemos predistorsión en RF, predistorsión en IF y predistorsión en banda base e inclusive predistorsión en datos[11]. La predistorsión de RF se suele implementar en modo analógico, mientras que la predistorsión en IF y banda base pueden implementarse de modo analógico[12] y digital.

La predistorsión analógica a lazo abierto presenta una arquitectura básica mostrada en la Fig. 2.5 y la función de predistorsión es implementada en usando un dispositivo no lineal.



$$y(t) = F\{P[x(t)]\} = k_0 x(t)$$

Fig. 2.5 Predistorsionador básico (*Open Loop Predistortion*).

Los esquemas más modernos de predistorsión tienden a ser adaptativos utilizando dispositivos reconfigurables para ajustar los coeficientes del predistorsionador. Típicamente se emplea una muestra de la potencia fuera de banda de la señal de salida del amplificador de potencia, o una señal de error, construida a partir de la propia señal de entrada y de salida, para dirigir el procedimiento de adaptación.

El procedimiento de adaptación del predistorsionador puede ser de manera continua y adaptativa o mediante entrenamiento.

El método adaptativo se realiza de manera continua durante toda la transmisión de la señal. De esta forma, los cambios en las características del amplificador de potencia o del canal en general pueden ser recogidos

---

[11] D'Andrea, Aldo N., Vincenzo Lottici, and Ruggero Reggiannini. "RF power amplifier linearization through amplitude and phase predistortion."Communications, IEEE Transactions on 44, no. 11 (1996): 1477-1484.

[12] Westesson, Eric, and L. Sundstrom. "A complex polynomial predistorter chip in CMOS for baseband or IF linearization of RF power amplifiers." In Circuits and Systems, 1999. ISCAS'99. Proceedings of the 1999 IEEE International Symposium on, vol. 1, pp. 206-209. IEEE, 1999.

por el algoritmo, conduciendo a un mejor ajuste para una infinidad de variaciones. Solamente aplica cuando la velocidad de convergencia del algoritmo sea mayor en comparación a los cambios del sistema.

Los métodos basados en entrenamiento, consiguen el ajuste de los parámetros del algoritmo mediante señales de entrenamiento que se transmiten al comienzo de la transmisión. Este puede parecer peor que el caso anterior, pero es válido para sistemas en los que se prevé que la no linealidad cambio poco, este método se ajusta muy bien al comportamiento de los amplificadores de potencia.

2.4.3.1 Métodos adaptativo

Los métodos adaptativos se llaman así por la forma en que consiguen la convergencia de sus parámetros. La mayoría de los métodos adaptativos se basan en una técnica que utiliza una tabla de datos (*Look Up Tables*). Estos datos son almacenados en la tabla, como muestras discretas provenientes de la aplicación de un algoritmo de modelado, por ejemplo series de Volterra. Empleando un modelo, generamos una correspondencia entre el valor de la señal de entrada y el valor de la señal que debe de pasar por el amplificador de manera que la salida tenga un comportamiento lineal.

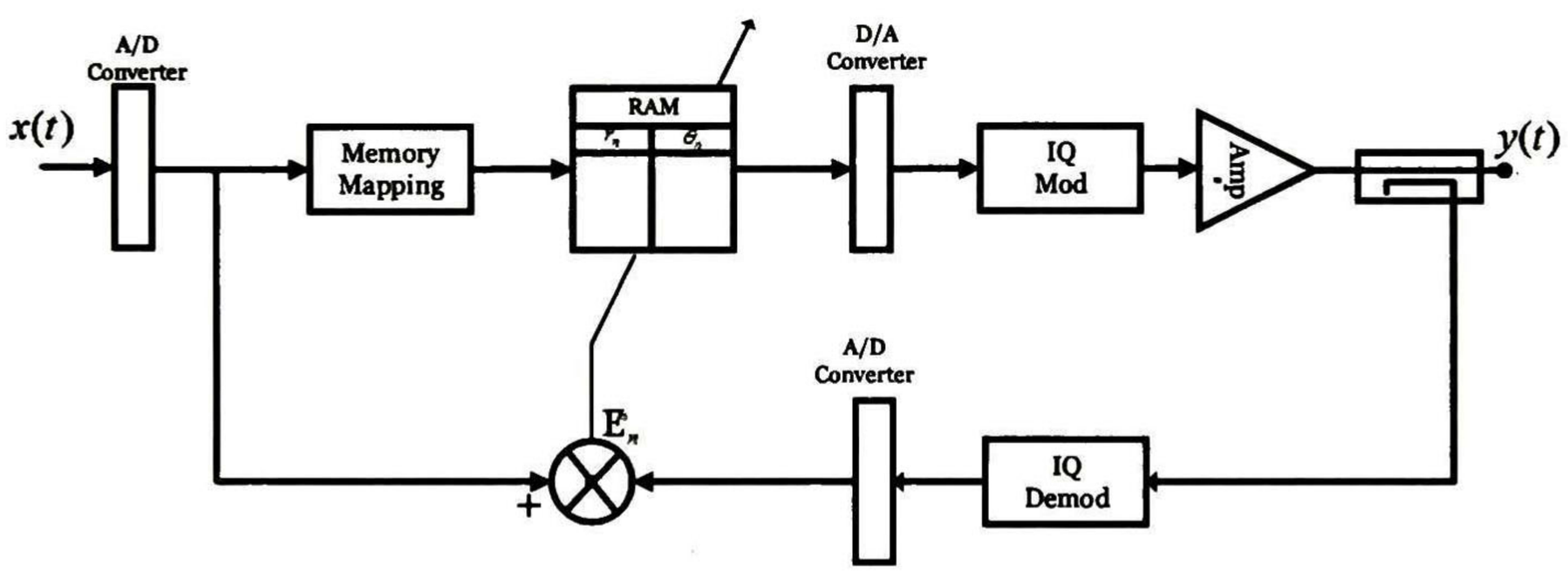


Fig. 2.6 Esquema de un predistorsionador adaptativo.

2.4.3.2 Predistorsión Digital

Predistorsión digital (DPD) es un método que consiste en emplear un algoritmo de procesamiento matemático que permita realizar la corrección de la respuesta de un sistema que no lineal. Comúnmente el procesamiento matemático es llevado por equipo de cómputo y un software matemático capaz de resolver las operaciones necesarias, pero debido al gran avance de dispositivos digitales como DSP, FPGA, y microprocesadores se han implementado diseños en hardware.

En la arquitectura mostrada en Fig. 2.7 se muestra un predistorsionador digital donde la función de predistorsión es sintetizada en un DSP utilizando LUTs. En esta arquitectura se añade un mecanismo de adaptación que actualiza los elementos almacenados en la tabla durante un intervalo de tiempo dedicado a este fin, después del cual el sistema es capaz de linealizar la respuesta del amplificador de potencia. El sistema puede asimilar las variaciones que puedan sufrir los parámetros del amplificador con el tiempo, siempre y cuando sean relativamente lentos comparados con el tiempo de convergencia del sistema adaptativo. Un sistema de predistorsión digital adaptativo tiene la forma básica mostrada en la Fig. 2.7. El bucle de retroalimentación provee al DSP de una muestra demodulada de la señal de salida.

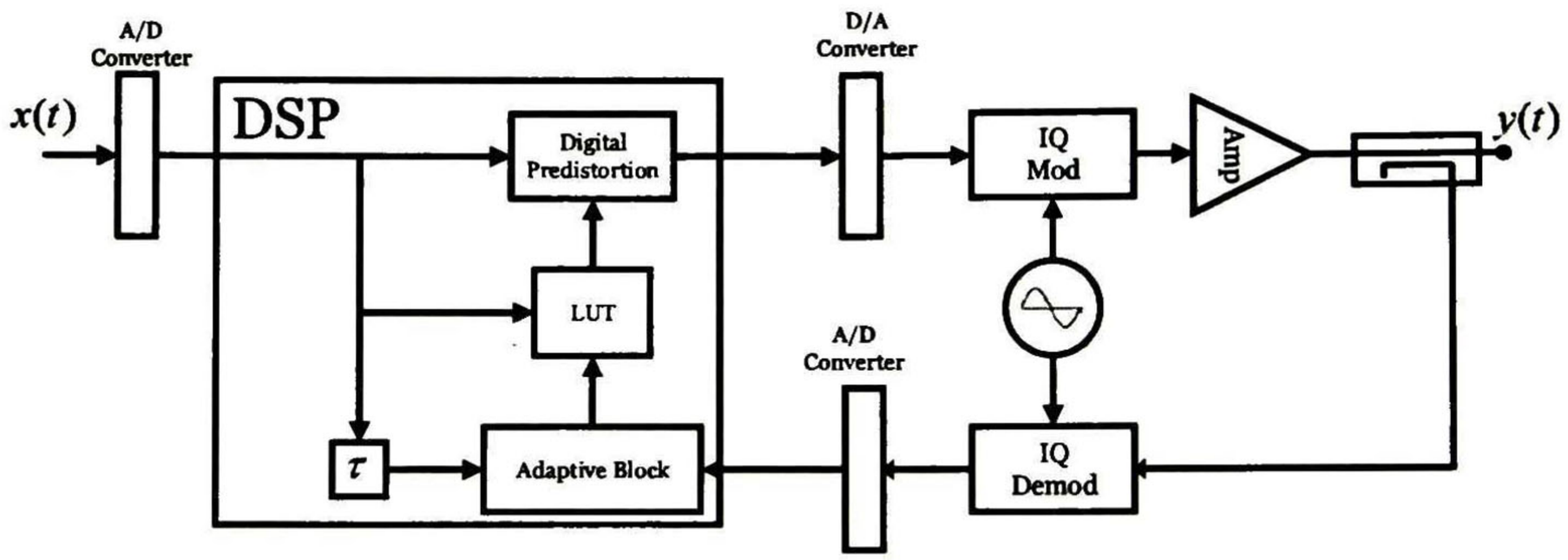


Fig. 2.7 Arquitectura básica de un predistorsionador digital.

Esta muestra y la señal de entrada son convertidas al dominio discreto mediante convertidores analógicos-digitales. Las señales discretas que se obtienen son además utilizadas para construir una señal de error a partir de la cual se actualiza la tabla que contiene la función de predistorsión. La señal de entrada es utilizada para leer la tabla, y como salida la tabla proporciona un valor de predistorsión que es aplicado a la señal de entrada. El valor predistorsionado es convertido al dominio analógico mediante un convertidor digital-analógico, posteriormente es modulado y amplificado.

Los predistorsionadores digitales han cobrado gran importancia en el tema de linealización de amplificadores de potencia, y se han buscado métodos alternativos para la implementación de estos diseños.

La aplicación de las series de Volterra, es un método viable de solución para el modelado de sistemas no lineales, ya que estas son una generalización de la convolución en la descripción de la respuesta temporal de un sistema lineal t-invariante. El primer término coincide precisamente con esta integral de convolución y caracteriza la componente lineal de la respuesta global del sistema no lineal. Las series de Volterra resultan conceptualmente apropiadas para modelar un sistema no lineal, ya que comprenden todos los efectos que se manifiestan

físicamente en el dispositivo. Las series de Volterra tienen la forma que se presenta en (2.1).

$$y(t)=\sum_{n=1}^{x}\int_{-\infty}^{\infty}\int_{-\infty}^{\infty}\cdots\int_{-\infty}^{\infty}h_n(\tau_1,\tau_2,\ldots,\tau_n)\prod_{k=1}^{n}x(t-\tau_n)\mathrm{d}\tau_k \qquad (2.1)$$

Otro método que se ha desarrollado para la solución de este problema, es conocido como memorias polinomiales[13].

Recientes investigaciones, han buscado alternativas de implementación de algoritmos que permitan obtener un mejor modelado de sistemas no lineales y debido a esto ha tomado gran importancia la computación neuronal. Las redes neuronales, son un campo de estudio que ha mostrado grandes ventajas y excelentes resultados en las aplicaciones en diversas materias. En aplicaciones de RF se ha convertido en un método atractivo de solución en aplicaciones de predicción de sistemas caóticos, modelado de amplificadores y predistorsión digital.

Las redes neuronales al presentar capacidad de procesamiento similar a las neuronas biológicas, manifiestan la capacidad de aprender de los eventos presentes, generalizar los problemas ante pequeñas variaciones y abstraer características de un sistema donde el resultado parecía carecer de relación alguna; estas características han permitido realizar desarrollos en diseños de predicción de señales[14], modelado de sistemas[15] y en predistorsión digital[16].

La predicción de series caóticas de tiempo con redes neuronales[17] es una solución común ante el problema que presentan los sistemas dinámicos. El enfoque de las redes neuronales para la predicción de series de tiempo es no paramétrico, por lo que no es necesario conocer información sobre el proceso que genera la señal. Una red neuronal recurrente, al presentar la capacidad de modelar los sistemas con memoria es capaz de predecir sistemas de este tipo con una buena aproximación. Sin embargo una de las limitaciones que presenta actualmente este método de predicción es

---

[13] Morgan, Dennis R., Zhengxiang Ma, Jaehyeong Kim, Michael G. Zierdt, and John Pastalan. "A generalized memory polynomial model for digital predistortion of RF power amplifiers." Signal Processing, IEEE Transactions on 54, no. 10 (2006): 3852-3860.

[14] Diaconescu, Eugen. "The use of NARX neural networks to predict chaotic time series." WSEAS Transactions on Computer Research 3.3 (2008): 182-191.

[15] Pantic, D., et al. "Inverse modeling of semiconductor manufacturing processes by neural networks." Microelectronics, 1995. Proceedings, 1995 20th International Conference on. Vol. 1. IEEE, 1995.

[16] Rawat, Meenakshi, Karun Rawat, and Fadhel M. Ghannouchi. "Adaptive digital predistortion of wireless power amplifiers/transmitters using dynamic real-valued focused time-delay line neural networks." Microwave Theory and Techniques, IEEE Transactions on 58.1 (2010): 95-104.

[17] Diaconescu, Eugen. "The use of NARX neural networks to predict chaotic time series." WSEAS Transactions on Computer Research 3, no. 3 (2008): 182-191.

causada por la cantidad de recursos utilizados por la red, y la complejidad que presentan los algoritmos de aprendizaje.

Realizar modelado de amplificadores de potencia[18] mediante las técnicas convencionales, es una tarea difícil de realizar debido a los efectos de memoria que presentan estos componentes, la aplicación de redes neuronales ha resultado ser un método de solución para el modelado de sistemas lineales y no lineales debido a la gran capacidad de aprendizaje que presentan ante los estímulos que se le presentan. Las redes neuronales recurrentes al tener la característica de modelar los efectos de memoria han sido aplicadas para la solución de este problema, y las arquitecturas más comunes para este fin son aquellas que presentan conexiones externas, por ejemplo las arquitecturas NARX[19], NARMAX[20] y RVTDNN[21].

2.4.3.2.1 Predistorsión digital con redes neuronales

Predistorsión digital hoy en día es una técnica usada para linealizar los amplificadores de potencia dado a la alta capacidad de corregir la distorsión y su aplicación puede ser tanto en software o hardware.

Las redes neuronales recurrentes han sido capaces de modelar los efectos de memoria de los amplificadores de potencia. En aplicaciones de predistorsión digital, es necesario anexar la característica de poder obtener la función inversa de la entrada-salida de la respuesta del amplificador. Debido a esto se han propuesto diferentes arquitecturas de redes neuronales para la solución del problema, destacando que el factor de mayor impacto es el método de aprendizaje empleado para la red, es decir si son definidos los parámetros correctos de entrenamiento, la red neuronal será capaz de resolver el problema aun cuando las entradas que se le presentan no sean idénticas a las empleadas por el entrenamiento debido a factores como ruido.

Una red entrenada correctamente presenta salidas consistentes a cualquier entrada, sin embargo, si los resultados obtenidos no son coherentes con una respuesta lógica correspondiente al sistema, puede que la red haya caído en un caso de sobre-entrenamiento y carezca de generalización de los datos. Algunos métodos para entrenar las redes

---

[18] O'Brien, Bill, John Dooley, and Thomas J. Brazil. "RF power amplifier behavioral modeling using a globally recurrent neural network." In *Microwave Symposium Digest, 2006. IEEE MTT-S International*, pp. 1089-1092. IEEE, 2006.

[19] Xie, Hang, Hao Tang, and Yu-He Liao. "Time series prediction based on NARX neural networks: An advanced approach." In Machine Learning and Cybernetics, 2009 International Conference on, vol. 3, pp. 1275-1279. IEEE, 2009.

[20] L. Bai, D. Coca, "Nonlinear predictive control based on NARMAX models", In Optimization of electrical and electronic equipment, 2008.

[21] Sandrin Ntoune Ntoune, R., Mohammed Bahoura, and Chan-Wang Park. "FPGA-implementation of pipelined neural network for power amplifier modeling." In New Circuits and Systems Conference (NEWCAS), 2012 IEEE 10th International, pp. 109-112. IEEE, 2012.

neuronales son "*Gauss-Newton*"; "*Levenberg-Marquardt*[22]" y "*gradientes conjugados*".

Se han hecho desarrollos con redes neuronales recurrentes como RVFTDNN[23] (*Real-Valued Fosused Time-Delay Neural Network*) que es una combinación de una red neuronal RVFFNN (*Real-Valued FeedForward Neural Network*) que tiene como ventaja la disponibilidad de entradas externas y una red recurrente FTDNN (*Focused Time-Delay Neural Network*) que modela los efectos de memoria, donde las salidas presentes dependen de los datos presentes y pasados. Esta red aprovecha las características de las redes neuronales mencionadas anteriormente para la aplicación de predistorsión digital, obteniendo una buena precisión en términos de densidad espectral de potencia. La red neuronal RVFTDNN fue implementada en matlab y la arquitectura se muestra en Fig. 2.8.

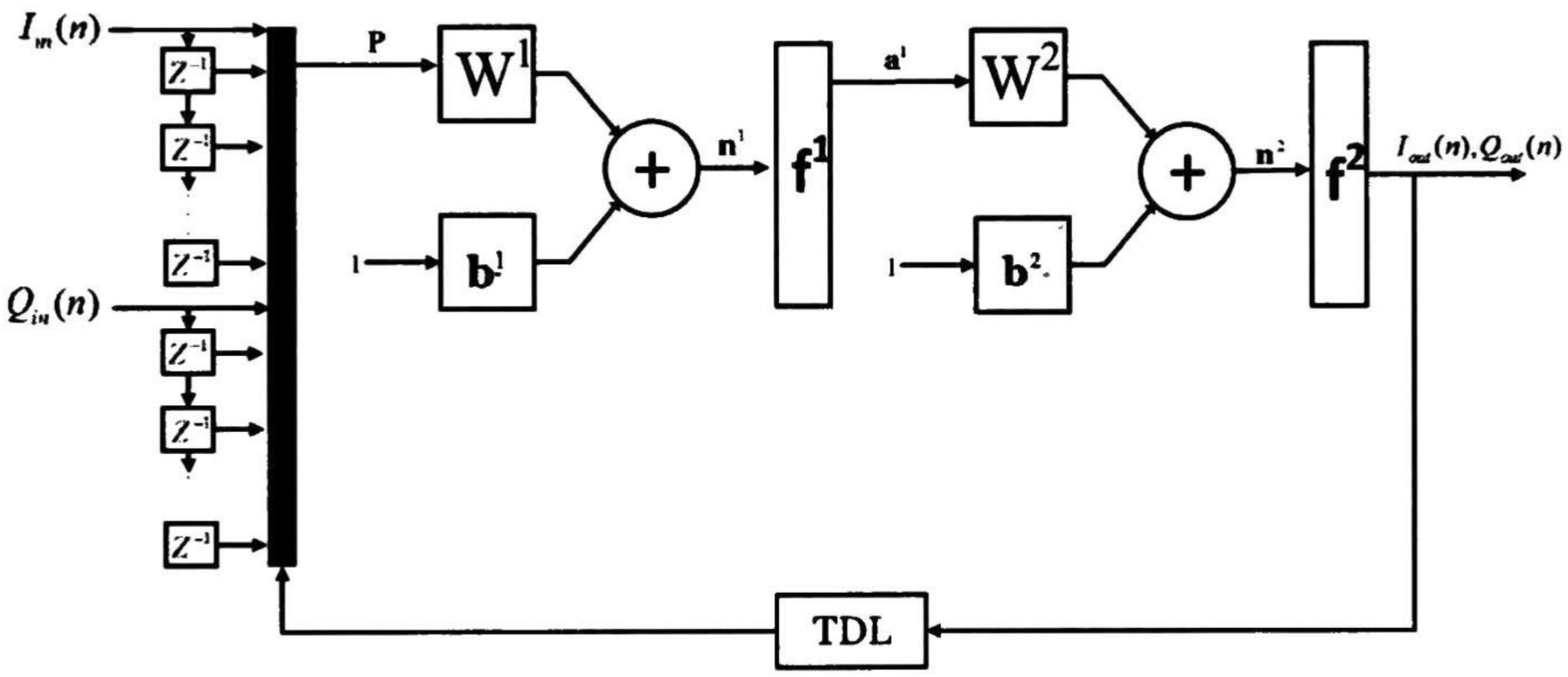


Fig. 2.8 Esquema de red neuronal RVFTDNN.

Realizando un revisión en la literatura sobre las diferentes configuraciones de redes neuronales, se han encontrada otras arquitecturas que son viables para la implementación de un predistorsionador digital.

La red neuronal NARX[24] (*Nonlinear AutoRegresive with eXogenous* inputs) es una arquitectura que cumple con los requisitos necesarios para

---

[22] Hagan, Martin T., and Mohammad B. Menhaj. "Training feedforward networks with the Marquardt algorithm." *Neural Networks, IEEE Transactions on* 5, no. 6 (1994): 989-993.

[23] Rawat, Meenakshi, Karun Rawat, and Fadhel M. Ghannouchi. "Adaptive digital predistortion of wireless power amplifiers/transmitters using dynamic real-valued focused time-delay line neural networks." Microwave Theory and Techniques, IEEE Transactions on 58.1 (2010): 95-104.

[24] L.M. Aguilar-Lobo, J.R. Loo-Yau, S. Ortega-Cisneros, "A Digital Predistortion Technique Based on a NARX Network to Linearize GaN Class F Power Amplifiers", *IEEE* 57th International Midwest Symposium on Circuits And Systems, August 2014.

el diseño, esta tiene entradas externas y además presenta buen modelado de efectos memoria. La velocidad de convergencia de la red NARX es mayor en comparación a otras arquitecturas.

La red NARX tiene una arquitectura no recurrente basada en una clase de sistema no lineal de tiempo discreto, y su representación matemática está dada por:

$$\tilde{y} = f\left[u(t-1),...u(t-du), y(t-1),...y(t-dy)\right] \tag{2.2}$$

donde $u(t)$ y $y(t)$ son las entradas y salidas en el tiempo t, du y dy representan las entradas y salidas retardadas y f es una función de activación no lineal.

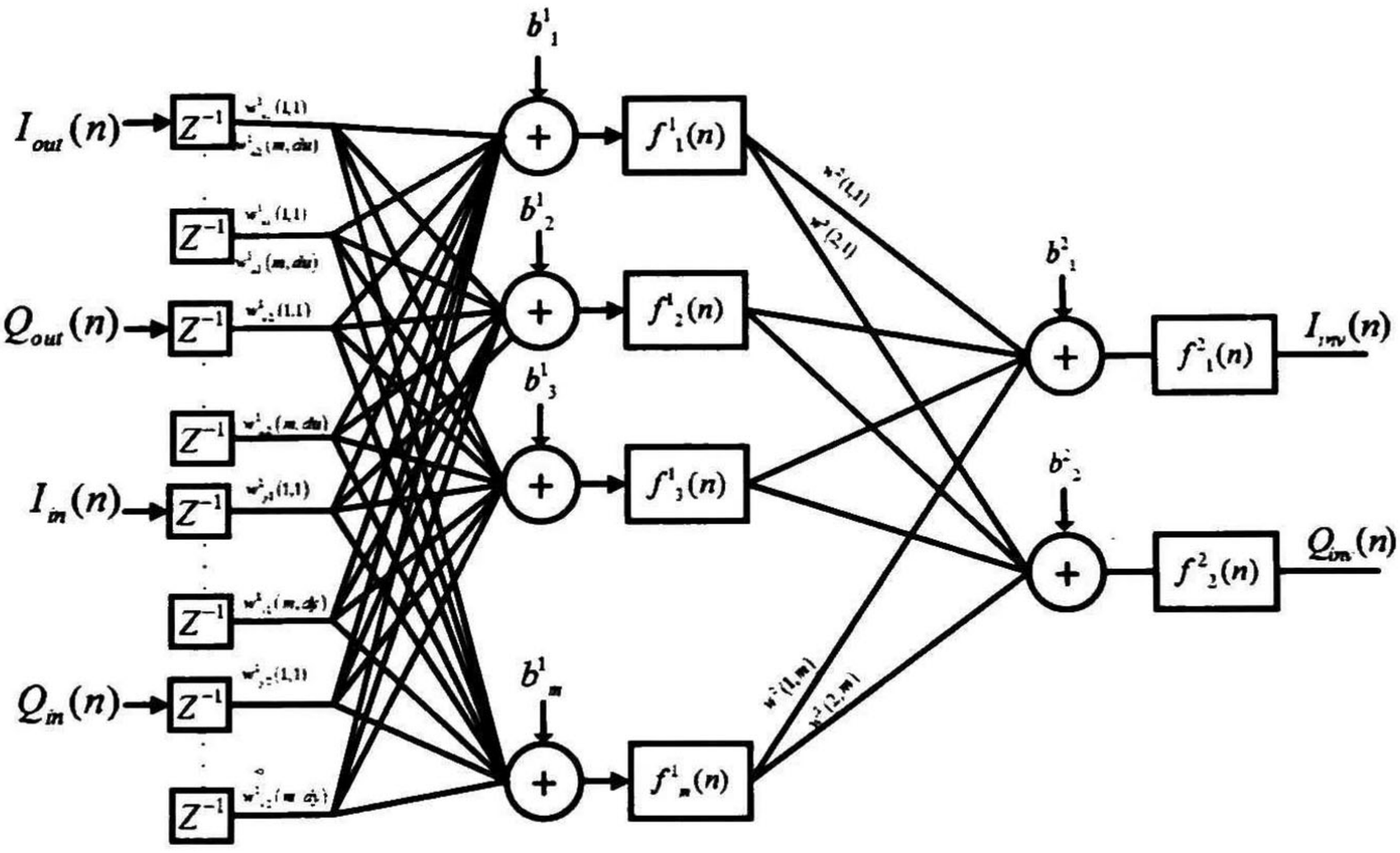


Fig. 2.9 Red neuronal NARX.

Esta arquitectura presenta buena correspondencia en los resultados obtenidos en el predistorsionador digital y tiene la capacidad de modelar los sistemas no lineales y los efectos de memoria de los amplificadores de potencia.

Los dispositivos reconfigurables han contribuido en el desarrollo de aplicaciones en plataformas portátiles, y en aplicaciones de predistorsión digital, los FPGA son comúnmente empleados gracias a la facilidad de reconfiguración. Algunas aplicaciones de redes neuronales en FPGA se

presentan en elaboración de controladores[25] en diseños de electrónica de potencia y modelado de amplificadores[26], sin embargo, cabe mencionar que los algoritmos de entrenamiento empleados aún siguen realizándose en software y la red neuronal en FPGA solo aprovecha los pesos de las conexiones y los valores de las polarizaciones que requiere la red para su funcionamiento.

Debido a la complejidad que implica la programación de una red neuronal en lenguaje de descripción de hardware, es comúnmente empleado un entorno de programación alternativo, como es el uso de una extensión de Matlab llamada *System Generator*[27], que es una forma de programación que permite trasladar una arquitectura diseñada en bloques a un archivo de descripción de hardware, empleado para la configuración del FPGA.

Diseños alternativos han empleado el uso de procesadores embebidos[28] como el *NIOS II* de Altera o *Microblazer* de Xilinx que permiten programar las redes neuronales en lenguaje de alto nivel, facilitando el manejo de datos y operaciones matemáticas necesarias para la implementación.

---

[25] Bastos, J. L., H. P. Figueroa, and A. Monti. "FPGA implementation of neural network-based controllers for power electronics applications." In Applied Power Electronics Conference and Exposition, 2006. APEC'06. 21th Annual IEEE, pp. 6-pp. IEEE, 2006.

[26] Ntouné, Roger Sandrin Ntouné, Mohammed Bahoura, and Chan-Wang Park. "Power Amplifier Behavioral Modeling by Neural Networks and Their Implementation on FPGA." In Vehicular Technology Conference (VTC Fall), 2012 IEEE, pp. 1-5. IEEE, 2012.

[27] Ownby, Matthew, and W. H. Mahmoud. "A design methodology for implementing DSP with Xilinx System Generator for Matlab." Insoutheastern Symposium On System Theory, vol. 35, pp. 404-408. 2003.

[28] Possignolo, Trapani. "Optimized joint NARX ANN-embedded processor design methodology." In *2009 16th IEEE International Conference on Electronics, Circuits and Systems-(ICECS 2009)*, pp. 499-502. 2009.

Capítulo 3

Redes neuronales

En esta sección se trata de describir las características que presentan las redes neuronales con respecto las redes biológicas, la nomenclatura estándar empleada para las redes neuronales artificiales, descripción del funcionamiento de una red neuronal y las arquitecturas más comunes. Se describen las funciones de activación y su posible campo de aplicación.

3.1 Introducción

LAS redes neuronales son modelos de comportamiento inteligente que pueden ser construidas de forma artificial basados en el sistema nervioso de los seres vivos. Están construidas por elementos que intentan simular las funciones más comunes de la neurona biológica donde aprenden de la experiencia, generalizan de ejemplos previos a ejemplos nuevos y abstraen las características principales de una serie de datos.

Aprender: Adquirir el conocimiento de una cosa por medio del estudio, ejercicio o la experiencia. Las neuronas pueden cambiar su comportamiento en función del entorno. Se les muestran un conjunto de entradas y ellas mismas se ajustan para producir unas salidas consistentes.

Generalizar: Las neuronas generalizan automáticamente debido a su propia estructura y naturaleza. Las redes pueden ofrecer dentro de un margen, respuestas correctas a entradas que presentan pequeñas variaciones debido a los efectos del ruido o distorsión.

Abstraer: Algunas neuronas son capaces de abstraer la esencia de un conjunto de entradas que aparentemente no presentan aspectos comunes o relativos.

Una red neuronal está compuesta por un conjunto de unidades procesadoras, donde las entradas de una neurona son las salidas de una neurona previa. Existen arquitecturas que se han propuesto a lo largo del tiempo para la solución de problemas particulares, como son procesamiento de voz, procesamiento de imagen o reconocimiento de patrones, aunque la aplicación de estos algoritmos es muy amplio.

Las redes neuronales presentan una arquitectura totalmente diferente de los ordenadores tradicionales de un único procesador. Por ejemplo las maquinas tradicionales basadas en el modelo de Von Neumann tienen un único elemento procesador, que realiza todos los cálculos ejecutando todas las instrucciones en secuencia, en una red neuronal parte del procesamiento se realiza en paralelo. Las redes neuronales son computacionalmente muy poderosas y para que sean capaces de resolver un problema, deben de ser entrenadas con un conjunto de entradas hasta que sean capaces de proporcionar salidas consistes, es decir que ha aprendido las características para resolver el problema. Sin embargo un buen entrenamiento se realiza cuando la red es capaz de generalizar el problema, y se le presentan entradas que no fueron empleadas para el entrenamiento obteniendo resultados favorables.

Existen redes neuronales de capa simple (*Single layer*) o múltiples capas (*multi-layer*), las redes de capa simple son empleadas para resolver problemas de baja complejidad y situaciones de frontera es decir donde la respuesta es linealmente separable por ejemplo en detección de bordes, mientras que las redes multicapa hacen frente a problemas complejos y son capaces de resolver funciones no lineales donde es difícil asociar entradas con salidas.

3.2 Neurona artificial

La neurona artificial fue diseñada para emular la neurona biológica y está constituida por: entradas, pesos, polarizaciones, unidad de procesamiento y función de activación. En esencia a la neurona se le aplica un conjunto de entradas que pueden representar salidas de neuronas previas o entradas externas, cada entrada es multiplicada por un "peso" ponderado correspondiente análogo al grado de conexión de la sinapsis. Todas las entradas ponderadas y el valor de la polarización se suman en la unidad de procesamiento para determinar el nivel de activación de la salida. Este tipo de modelo de neurona artificial ignora muchas de las características de las neuronas biológicas. Entre ellas destaca la omisión de retardo y de sincronismo en la generación de la salida, a pesar de estas limitaciones las redes neuronales construidas con

este tipo de neurona artificial presentan cualidades y atributos con cierta similitud a la de los sistemas biológicos.

3.2.1 Neurona de entrada simple

Este tipo de neurona solo cuenta con una entrada p, y es multiplicada por el valor ponderado del peso w y es adicionado un valor de polarización b. El resultado obtenido es pasado a través de una función de transferencia f que definirá el nivel de activación de la salida.

La función que rige el funcionamiento de esta neurona está dada por:

$$a = (wp + b) \tag{3.1}$$

y la Fig. 3.1 muestra la representación de la neurona.

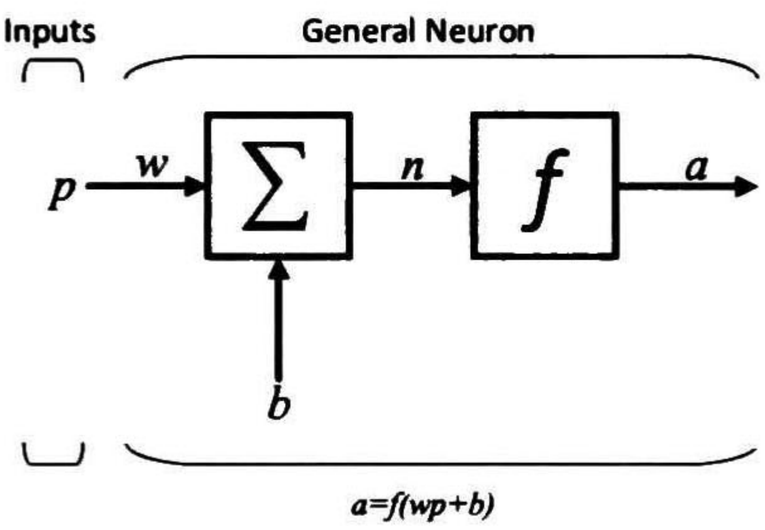


Fig. 3.1 Neurona con entrada simple.

3.2.2 Función de activación

La función de transferencia puede ser una función lineal o no lineal. Y la función de transferencia comúnmente es cambiada para satisfacer alguna especificación del problema que la neurona intenta resolver. La función de activación es la que define el nivel de excitación de la salida de la neurona y las funciones más comunes son: función escalón, función lineal, sigmoidea y tangente sigmoidea o también conocida como tangente hiperbólica.

La función escalón o *hard limit functions*, ajusta la salida de la neurona a cero si el argumento de entrada es menor a cero o a uno si la entrada es mayor igual a uno. Este tipo de función de activación es empleada para clasificar entradas en dos categorías. La Fig. 3.2 muestra la salida de una entrada escalón ideal y la obtenida con una neurona.

La salida de la función escalón esta descrita en (3.2).

$$a = 0, n < 0$$
$$a = 1, n \geq 0 \qquad (3.2)$$

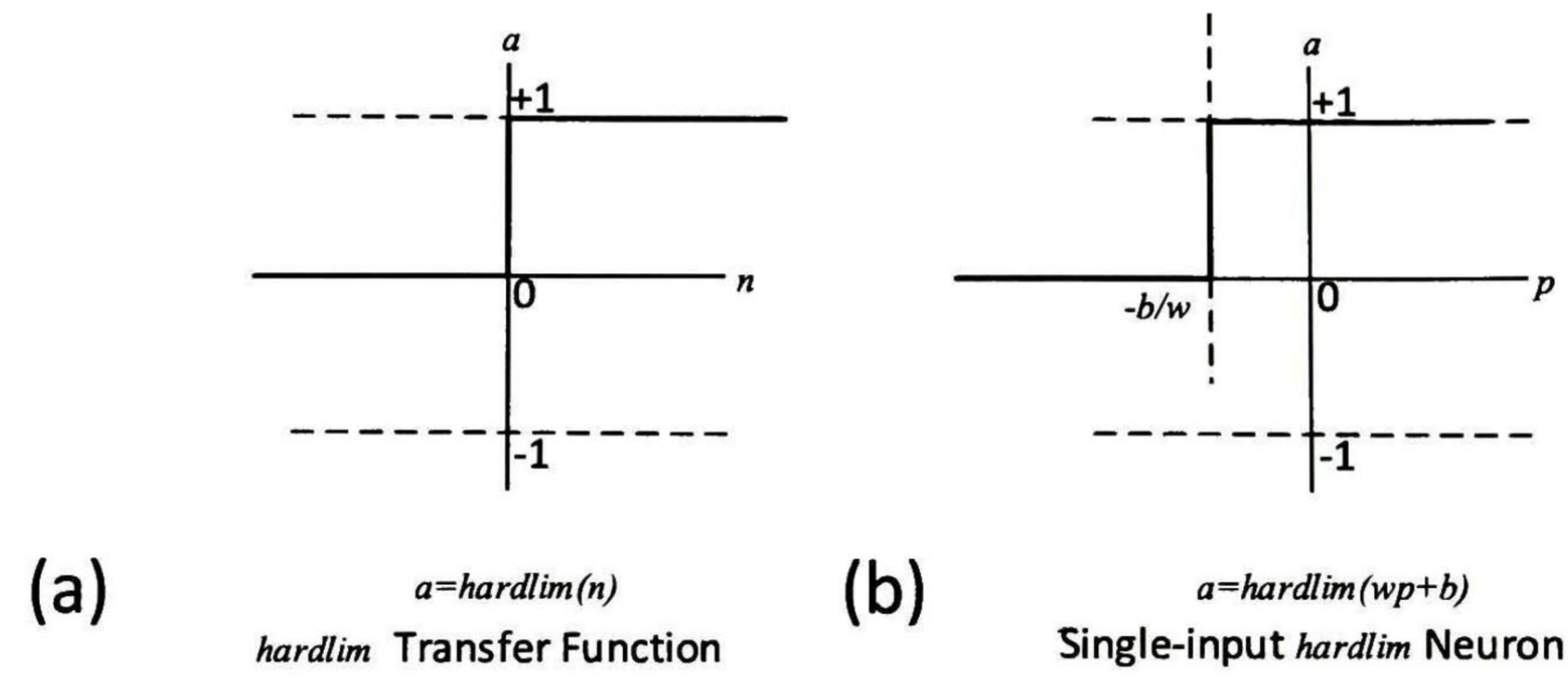


Fig. 3.2 Función escalón.

Cuando la aplicación de la red neurona tiene la finalidad de operación con condiciones de frontera, es comúnmente emplear un función lineal o "*linear transfer*" donde la salida es igual a entrada. Esta función aplicada en una red neuronal el peso y polarización provocan un desplazamiento en el eje horizontal. La función de salida se muestra en Fig. 3.3.

$$a = n \qquad (3.3)$$

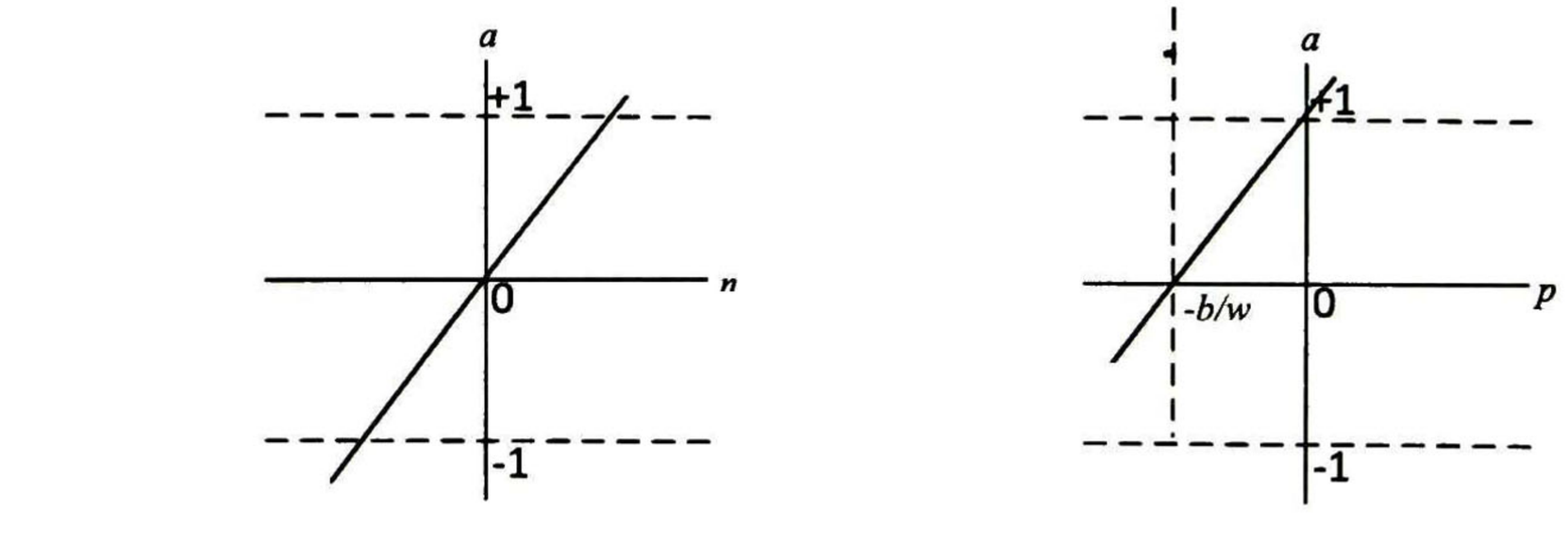


Fig. 3.3 Función lineal.

Cuando el objetivo del problema a resolver presenta características no lineales, el común emplear una función sigmoidea o "*log-sigmoid*". esta función no lineal comprime la salida en un rango de cero a uno y es empleada en redes multicapa como la "*backpropagation*" donde se

requiere de una función de activación que sea diferenciable. El comportamiento de esta función se muestra en Fig. 3.4.

La función sigmoidea está dada por (3.4).

$$a = \frac{1}{1+e^{-n}} \tag{3.4}$$

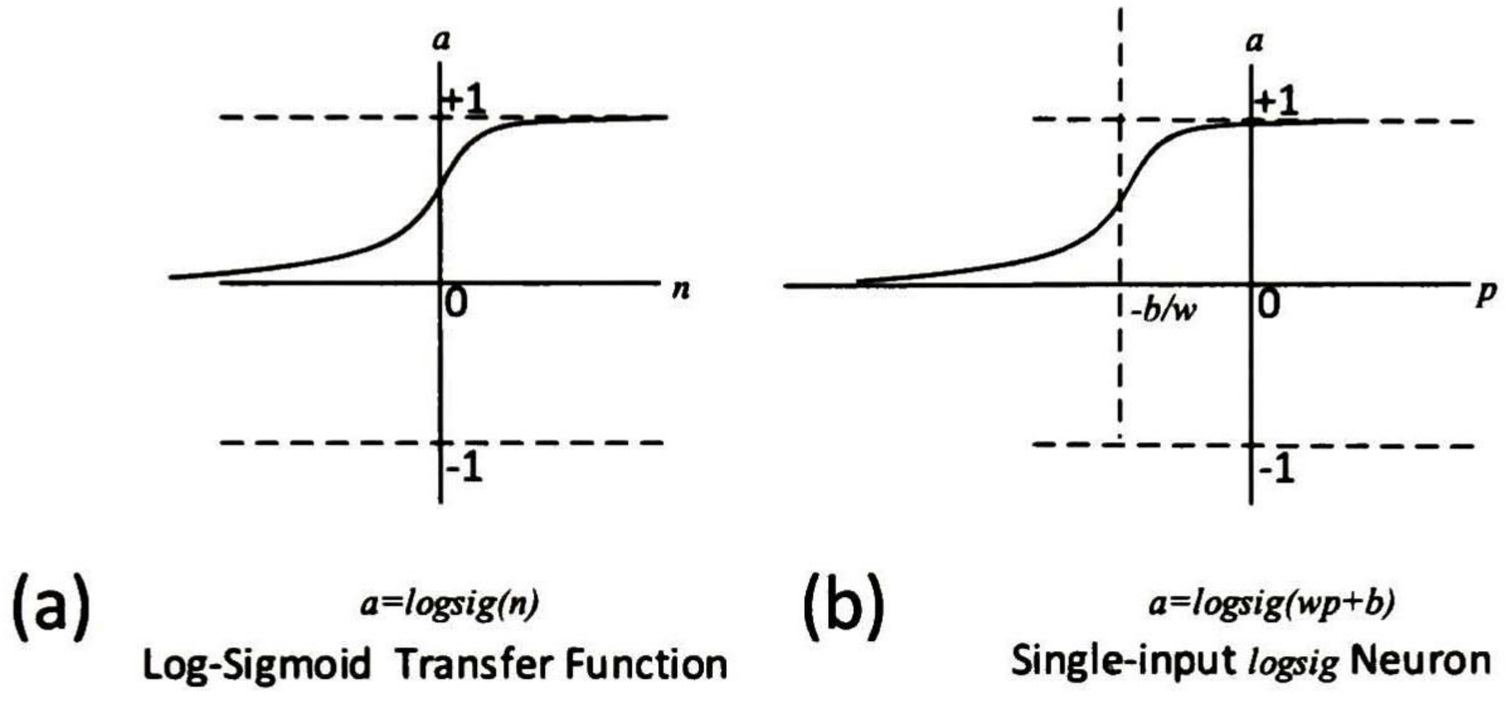


Fig. 3.4 Función sigmoidea.

La tangente sigmoidea o *tansig* es otra función que también es empleada para sistemas no lineales, su característica principal es que su salida puede oscilar entre valores positivos y negativos en una rango de menos uno y uno. Esta función es empleada en redes multicapa.

La función de salida está dada por la ecuación:

$$a = \frac{e^{n} - e^{-n}}{e^{n} + e^{-n}} \tag{3.5}$$

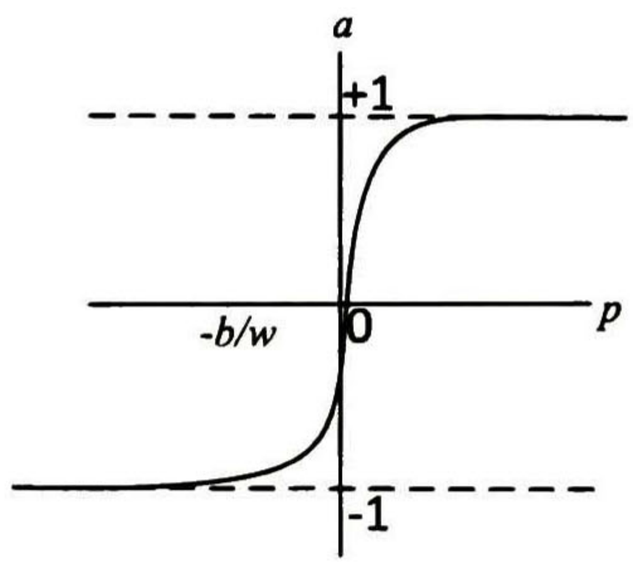


Fig. 3.5 Función tangente sigmoidea o tangente hiperbólica.

3.2.3 Neurona de múltiples entradas

Comúnmente una neurona tiene más de una entrada. Si se consideran R entradas como la que se muestra en Fig. 3.6, entonces para cada entrada $(p_1, p_2, \ldots p_R)$, se tiene un peso ponderado correspondiente a la conexión $(w_{1,1}, w_{1,2}, \ldots w_{1,R})$ de una matriz de pesos **W**.

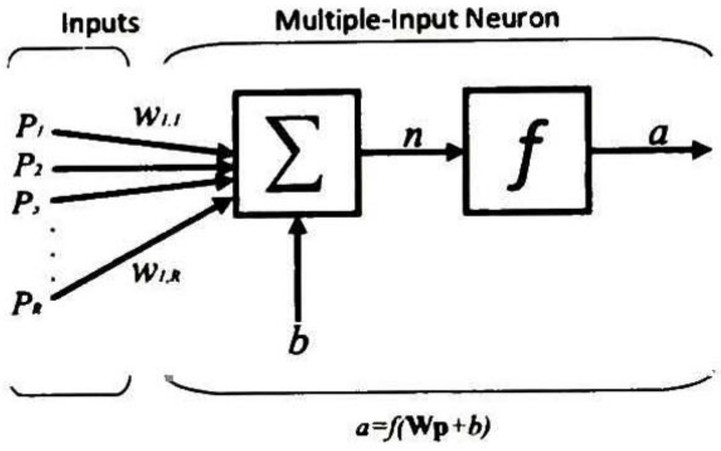


Fig. 3.6 Neurona de múltiples entradas.

Para la nomenclatura empleada en los pesos, el primer subíndice que lo integra indica la neurona a la que pertenece mientras que el segundo subíndice indica la fuente de entrada.

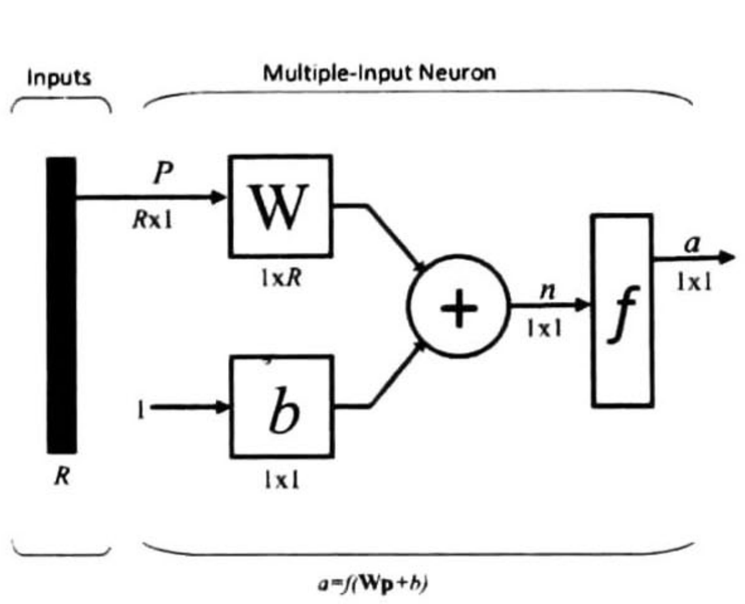


Fig. 3.7 Notación abreviada para neurona con R entradas.

Como se muestra en Fig. 3.7, el vector de entrada p es representado por un bloque solido vertical. Las dimensiones de p quedan definidas por $R \times 1$ donde la entrada es un vector simple de R elementos. Esas entradas van a una matriz de peso W, con R columnas peso solo una fila para el caso de la neurona simple. La constante 1 es multiplicada por el valor escalar de la polarización b. Las entradas son multiplicadas por los pesos, los cuales son sumados con la polarización y la salida es pasada por la función de activación para determinar la salida de la neurona que es un dato escalar.

3.3 Arquitecturas de redes neuronales

Comúnmente una neurona con varias entradas no es suficiente, por lo que se realizan conexión de un número mayor de neuronas que procesen en paralelo, a este concepto es conocido como capa del inglés "*layer*".

3.3.1 Red neuronal de una capa

Una capa sencilla de S neuronas se muestra en Fig. 3.8, donde cada una de las entradas R es conectada a cada neurona que tiene una matriz de pesos de S filas.

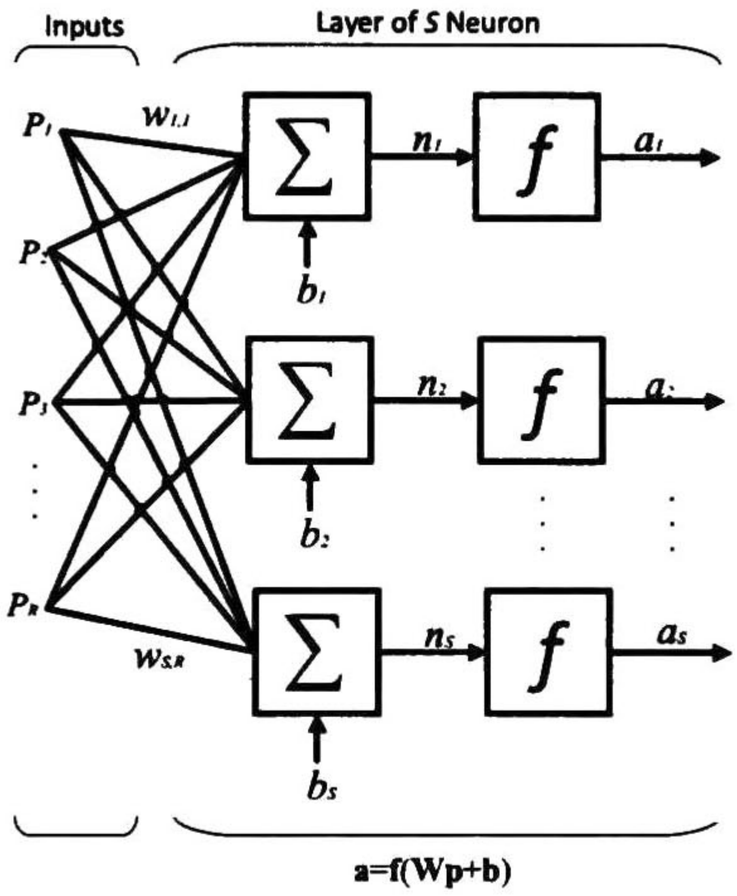


Fig. 3.8 Capa sencilla de S neuronas.

Es común que el número de entradas de las capa es diferente al número de neuronas, así como también todas las funciones de transferencia de las neuronas pueden ser definidas de forma individual.

Los elementos del vector de entrada de la matriz de pesos W se muestra en (3.6), donde el número de filas indica las neuronas y el número de columnas indica las entradas asociadas. Empleando expresiones matriciales, se puede dibujar una red con notación abreviada como se muestra en Fig. 3.9.

$$\mathbf{W} = \begin{bmatrix} w_{1,1} & w_{1,2} & \cdots & w_{1,R} \\ w_{2,1} & w_{2,2} & \cdots & w_{2,R} \\ \vdots & \vdots & & \vdots \\ w_{S,1} & w_{S,2} & \cdots & w_{S,R} \end{bmatrix} \tag{3.6}$$

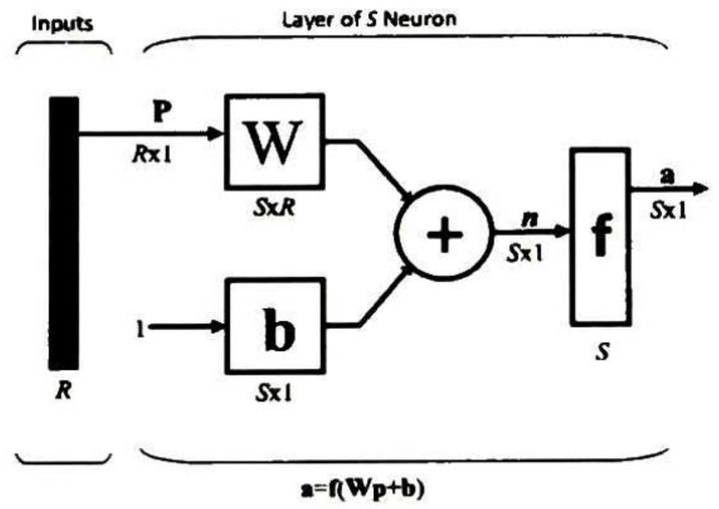


Fig. 3.9 Notación abreviada de capa sencilla de *S* neuronas.

3.3.2 Red neuronal multicapa

Las redes multicapa son computacionalmente superiores y puede aproximar una función continua hasta un nivel deseado, estas constan de una capa de entrada, una capa de salida y una o más capas ocultas, dichas capas se unen de forma secuencial hacia delante.

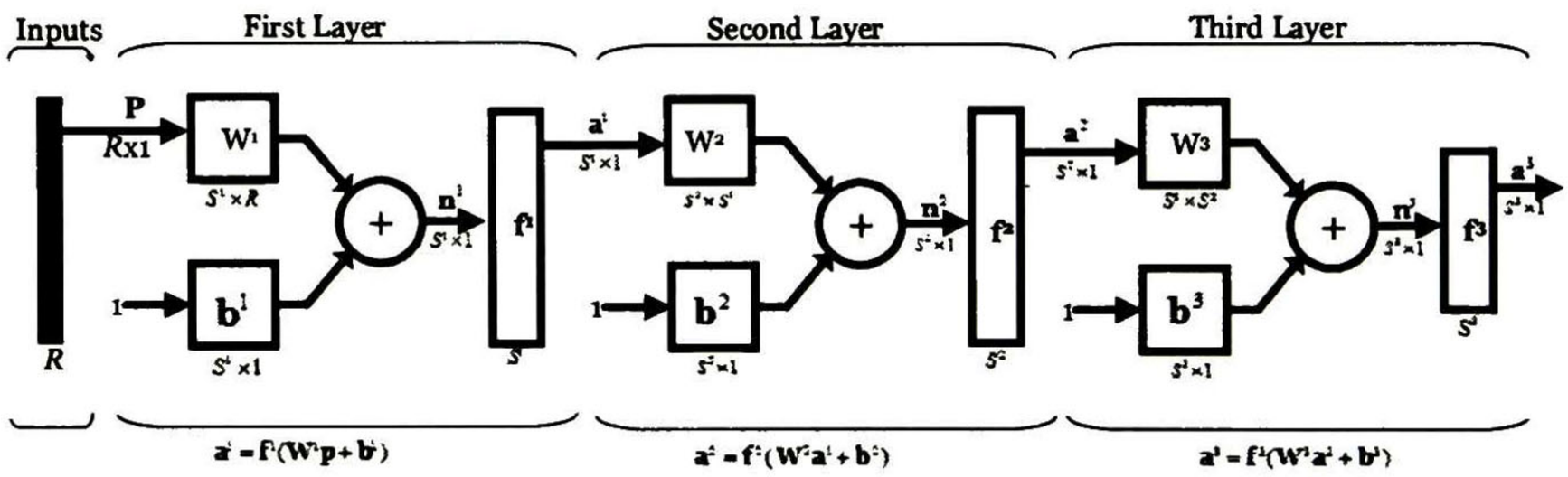


Fig. 3.10 Notación abreviada de red multicapa.

Es difícil definir el número óptimo de neuronas en la capa oculta y este es un problema directo en el consumo de recursos y área para diseños en hardware.

Para el entrenamiento emplean un algoritmo denominado de retropropagacion del inglés *backpropagation*, para ello se requiere que la red neuronal disponga parte de la información a procesar, es decir las entradas y salidas.

3.3.3 Red neuronal recurrente

Para describir una red neuronal recurrente, es necesario introducir un nuevo componente llamado bloque de retardo. En este bloque la salida es retardada por una muestra de tiempo, asumiendo que los datos se presentan en tiempo discreto.

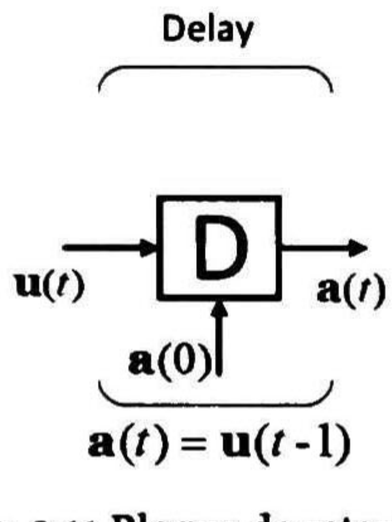


Fig. 3.11 Bloque de retardo.

Una red recurrente es una red con realimentación, donde algunas de las salidas son conectadas a la entrada, algo muy diferente de las redes mostradas anteriormente. Las redes recurrentes son potencialmente más poderosas que las redes de alimentación hacia adelante ya que pueden modelar efectos de memoria.

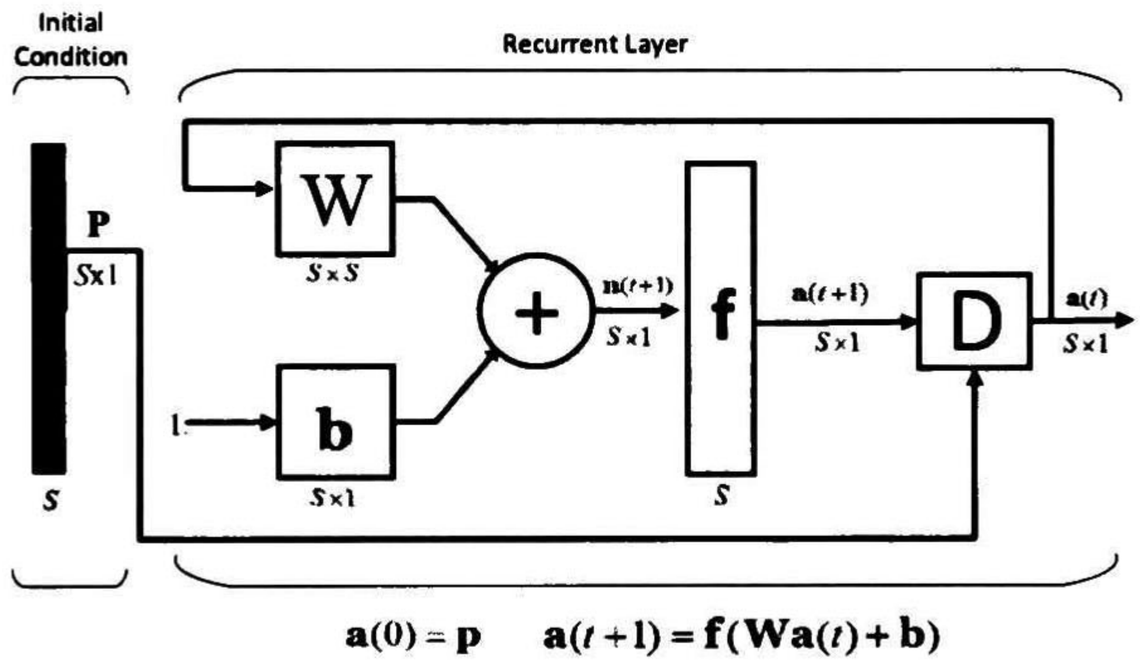


Fig. 3.12 Red neuronal recurrente.

3.4 Entrenamiento de redes neuronales

Una de las principales características de las redes neuronales es su capacidad de aprendizaje. El objetivo del entrenamiento es conseguir una aplicación determinada para que un conjunto de entradas produzca el conjunto de salidas deseadas o con mínimas variaciones. El proceso de entrenamiento consiste en la aplicación secuencial de diferentes conjuntos o vectores de entrada para que se ajusten los pesos de las interconexiones según el procedimiento predeterminado. Durante el proceso de entrenamiento los pesos convergen gradualmente hacia los valores que hacen que cada entrada produzca el vector de salida deseado.

Los algoritmos de entrenamiento de las redes neuronales se pueden clasificar en dos grupos: supervisado y no supervisado.

Entrenamiento supervisado: Estos algoritmos requieren del emparejamiento del vector de entrada con su correspondiente vector de salida. El entrenamiento consiste en presentar un vector de entrada a la red, calcular la salida de ésta, compararla con la salida deseada, y el error o diferencia resultante se utiliza para realimentar la red y cambiar los pesos de acuerdo con un algoritmo que tiende a minimizar el error.

Las parejas de vectores del conjunto de entrenamiento se aplican secuencialmente y de forma cíclica. Se calcula el error y el ajuste de los pesos por cada pareja hasta que el error para el conjunto de entrenamiento entero sea un valor pequeño y aceptable.

Entrenamiento no supervisado: Los sistemas neuronales con entrenamiento supervisado han tenido éxito en muchas aplicaciones y sin embargo tienen muchas críticas debido a que desde el punto de vista biológico no son muy lógicos. Los sistemas no supervisados son modelos de aprendizaje donde no se requieren de un vector de salidas deseadas y por tanto no se realizan comparaciones entre las salidas reales y las esperadas. Los vectores de entrenamiento consisten únicamente en los vectores de entrada. El algoritmo de entrenamiento modifica los pesos de la red de forma que produzca vectores de salida consistentes. El proceso de entrenamiento extrae las propiedades estadísticas del conjunto de vectores de entrenamiento y agrupa en clases los vectores similares.

3.5 Aplicaciones de las redes neuronales

Las características especiales de la computacional neuronal permiten que sea utilizada esta técnica de cálculo en una extensa variedad de aplicaciones. Alguna de las aplicaciones de estos sistemas son mostrados a continuación.

Conversión texto a voz: La ventaja que ofrece la computación neuronal frente a las tecnologías tradicionales en la conversión texto-voz es la propiedad de eliminar la necesidad de programar un complejo conjunto de reglas de pronunciación en el ordenador.

Compresión de imágenes: La compresión de imágenes es la transformación de los datos de una imagen a una representación diferente que requiere menos memoria y que se pueda reconstruir una imagen sin cambios perceptibles.

Reconocimiento de patrones e imágenes: Es la aplicación que se le da al clasificar los objetos detectados, o bien empleado en la detección de objetos en inspecciones industriales.

Procesado de señal: En este tipo de aplicación existen tres clases diferentes de procesado de señal que han sido objetos de las redes neuronales como son la predicción, el modelando de un sistema y el filtrado de ruido.

Predicción: Existen fenómenos de los que se puede conocer su comportamiento a través de una serie temporal de datos o valores. Las redes *backpropagation* han superada considerablemente a los métodos de predicción por polinomicos y lineales convencionales para las series temporales caóticas.

Modelado de Sistemas: Los sistemas lineales son caracterizados por la función de transferencias que es una expresión analítica entre la variable de la salida y una variable independiente y sus derivadas. Las redes neuronales son capaces de aprender una función de trasferencia y comportarse correctamente como el sistema lineal que está modelando.

Filtro de ruido: Las ANNs pueden ser utilizadas para eliminar el ruido de una señal. Estas redes son capaces de mantener en un alto grado las estructuras y valores de los filtros tradicionales.

Modelos económicos y financieros: Unas de las aplicaciones de importancia es el modelado y pronóstico de sistemas financieros.

ServoControl: Un problema difícil en el control de un complejo sistema de servomecanismos es encontrar un método de cálculo computacional aceptable para compensar las variaciones físicas que se producen en el sistema. Ente los inconvenientes destaca la imposibilidad en algunos casos de medir con exactitud las variaciones producidas y el excesivo tiempo de cálculo requerido para la obtención de la solución matemática. Existen diferentes redes neuronales que han sido entrenadas para reproducir o predecir el error que se produce en la posición final de un robot. Este error se combina con la posición deseada para proveer una posición adaptativa de corrección y mejorar la exactitud de la posición final.

Capítulo 4

Diseño de la Red NARX

En este capítulo se presenta detalladamente la arquitectura para la implementación de un predistorsionador digital empleando redes neuronales, incluyendo una descripción detallada de los elementos que lo constituyen, se analizan diferentes alternativas de implementación de funciones matemáticas y formatos numéricos propuestos.

4.1 Introducción

En este capítulo se propone la arquitectura de un predistorsionador digital realizado con una red neuronal NARX en un dispositivo reconfigurable FPGA de Xilinx, el sistema debe de ser capaz de transmitir los datos procesados por la red neuronal a un equipo de cómputo para su posterior simulación y validación de resultados. La herramienta de desarrollo empleada es una *Virtex-6 FPGA ML605 Evaluation Kit*[29] y la herramienta para su configuración es el software *ISE Design Suite de Xilinx*[30], que permite realizar diseños empleando lenguajes como: *VHDL* (*Very High Speed Integrated Circuits Description Language*) que es un lenguaje de descripción de hardware para circuitos integrados de muy alta velocidad, Verilog o diseños en forma esquemática. Esta herramienta cuenta con bloques *IP* (*Intellectual Property*) que son

[29] www.xilinx.com/support/documentation/boards_and_kits/ug534.pdf

[30] www.xilinx.com/support/download/index.html/content/xilinx/en/downloadNav/design-tools.html

algoritmos especializados o funciones matemáticas optimizadas propuestas por la plataforma de desarrollo.

4.2 FPGA

Se emplea un dispositivo reconfigurable FPGA de Xilinx para la implementación del predistorsionador digital, y mediante lenguaje de descripción de hardware, se realizan los algoritmos necesarios para el funcionamiento del diseño propuesto.

4.2.1 Tarjeta Xilinx ML605

El dispositivo reconfigurable que se utilizó para la implementación del sistema se muestra en la Fig. 4.1.

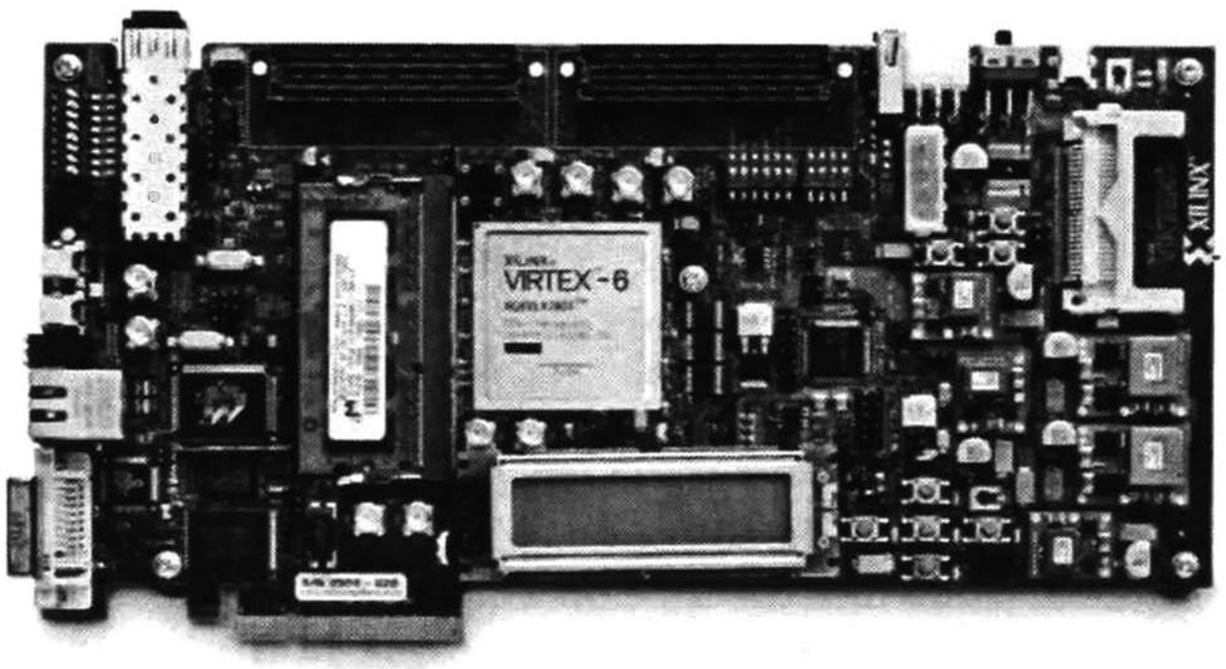


Fig. 4.1 Tarjeta de evaluación ML605.

Las características de la tarjeta de evaluación se mencionan a continuación:

- FPGA Virtex-6 modelo XC6VLX240T.
- Memoria RAM DDR3 de 512MB SODIMM.
- Memoria flash de 128 MB.
- USB JTAG.
- Sistema generador de reloj.
- Reloj diferencial de 200 MHz.
- Conector para oscilador de 2.5V.
- Interfaz Ethernet *PHY SGMII*.
- Conectividad PCIe.

- Adaptador integrado USB-UART.
- Controlador USB.
- Códec DVI.
- Bus I²C.
- EEPROM I²C de 1KB.
- LEDs indicadores.
- E/S de usuario.
- Interruptores de configuración.

4.3 Virtex-6

Algunas de las características importantes para la familia FPGA Virtex-6[31], se muestran a continuación:

- *LUTs* de 6 entradas reales con opción de funcionar como 2 *LUTs* de 5 entradas.
- *LUT-Flip-Flop* para aplicaciones que requiere gran cantidad de registros.
- Opción de emplear *LUTs* como memoria *RAM* distribuida de 32 y 64 bits.
- Potentes manejadores de reloj en modo mixto (*MMCM: Mixed Mode Clock Manager*) que proporcionan un retardo de cero en las líneas de reloj, síntesis de frecuencia, desplazamiento de fase y división de frecuencia
- Bloques de memoria de 32 Kb que pueden funcionar como memorias de puerto simple o doble.
- Contiene *Slices* avanzados *DSP48E1*, los cuales proporcionan capacidad para implementar multiplicadores y acumuladores en hardware.

Como se describen las características en la **Tabla 4.1**, el FPGA cuenta con un gran número de componentes lógicos para la implementación de diseños complejos como los diseños de predistorsión digital.

[31] http://www.xilinx.com/support/documentation/data_sheets/ds150.pdf

Tabla 4.1 Tabla de componentes lógicos del FPGA XC6VLX240T.

Componente lógico	Cantidad
Celdas lógicas	241,152
Slices	37,680
Máxima RAM Distribuida (Kb)	3,650
Slices DSP48E1	768
Máxima RAM en bloque (Kb)	14,976
MMCMs	12
Interfaces PCIe	2
MACs Ethernet	4
Transceptores GTX	24
Bancos de E/S	18
Pines máximos de E/S	720

Como se muestran las características en la **Tabla 4.1**, el FPGA cuenta con un gran número de componentes lógicos los cuales son una buena opcion para la implementación de diseños complejos.

4.4 Red neuronal NARX

La red NARX o *Nonlinear AutoRegressive network with eXogenous inputs*, es un tipo de red dinámica, con conexiones de realimentación que encierra varias capas de la red. El modelo de NARX está basado en el modelo linear ARX, el cual comúnmente es empleado para modelado de series de tiempo. La ecuación que rige la red neuronal está dado por:

$$\tilde{y} = f\left[u(t-1),...u(t-du),y(t-1),...y(t-dy)\right] \qquad (4.1)$$

donde $u(t)$ y $y(t)$ son las entradas y salida en el tiempo t, du y dy representan las entradas y salidas retardadas y f es una función de activación no lineal.

Existen diversas aplicaciones de la red NARX, entre ellas pueden ser usadas para predicción de la señal de entrada, filtrado no lineal para obtener una señal de salida libre de ruido y como último es el modelado no lineal de sistemas dinámicos.

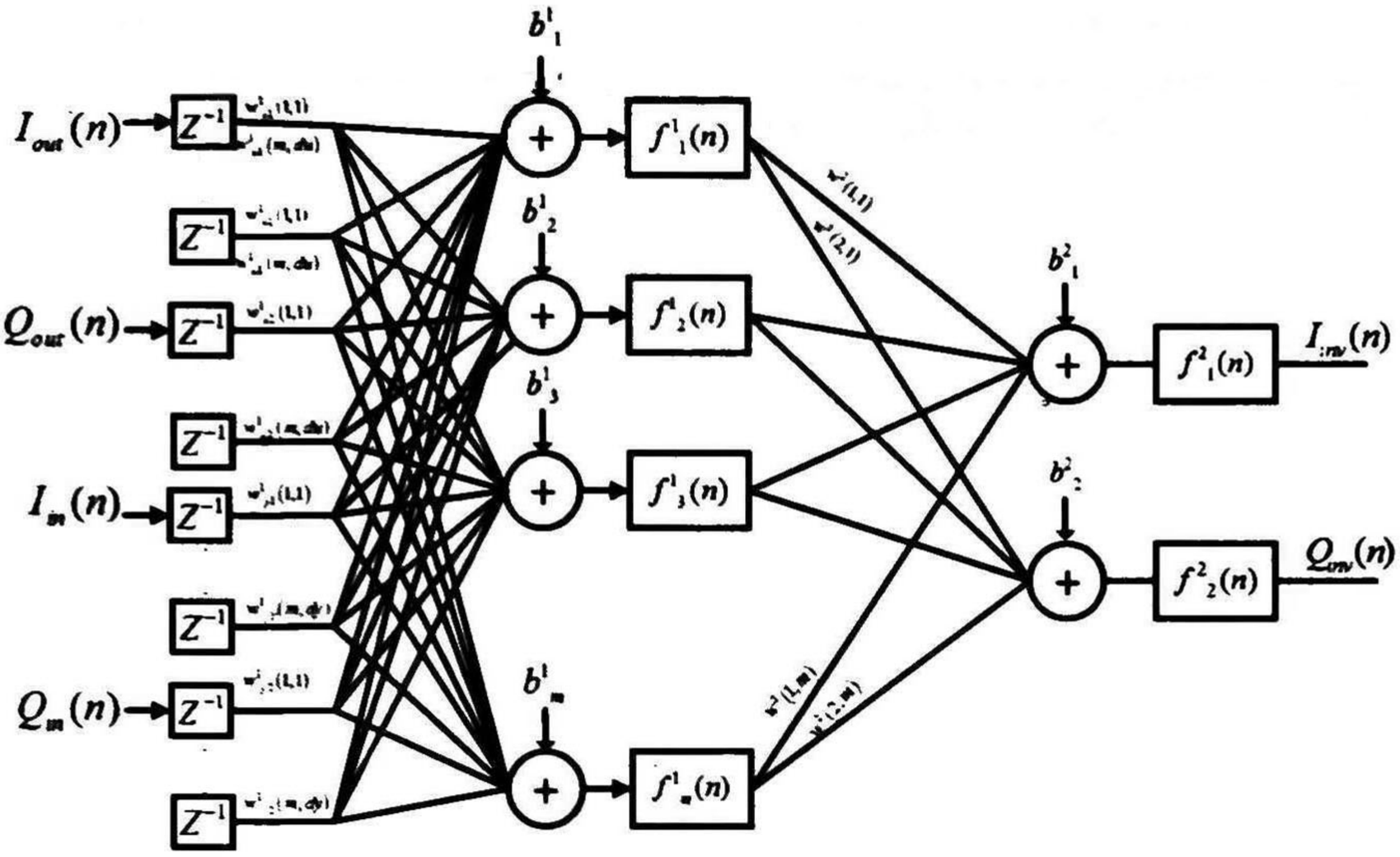


Fig. 4.2 Red neuronal NARX.

La red NARX cuenta con tres capas que son: la capa de entrada que engloba las entradas externas del sistema y el bloque de retardos de la red, la capa oculta que contiene un número de neuronas definidas por el usuario que están descritas como parte de las características iniciales de configuración de la arquitectura, esta capa contiene un función de activación no lineal que de forma predeterminada es la tangente hiperbólica y por último la capa de salida que contiene un número de neuronas que está ligado a la cantidad de salidas deseadas del sistema y la función de activación de estas neuronas es una respuesta lineal.

La red neuronal NARX, puede ser entrenada con dos configuraciones diferentes.

- Se puede considerar que la salida de la red NARX sea una estimación de la producción de algún sistema dinámico no lineal que se está tratando de modelar. La salida alimenta a la entrada de la red neuronal como parte de la arquitectura NARX estándar llamada arquitectura paralela.

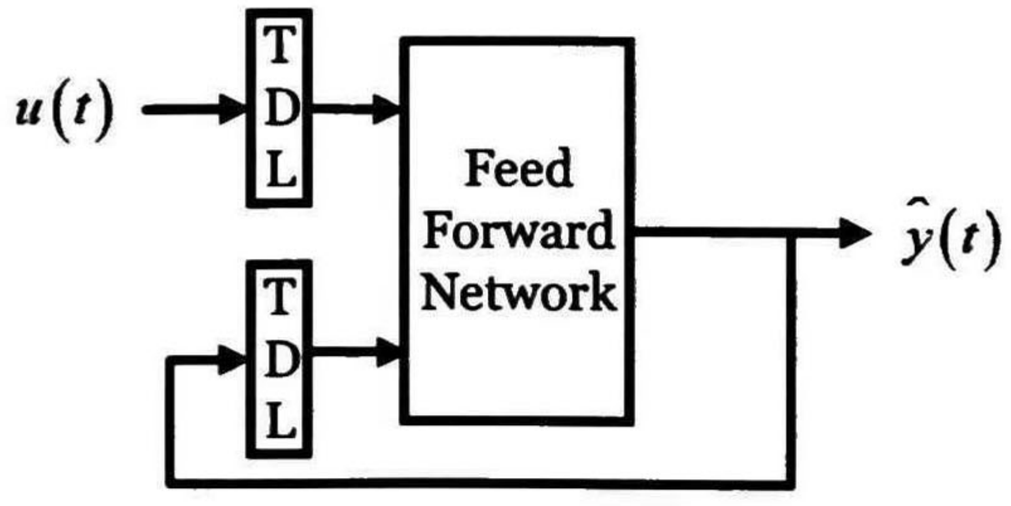


Fig. 4.3 Arquitectura Paralela.

- Cuando la salida real está disponible durante el entrenamiento de la red, se puede crear una arquitectura en serie-paralelo, en el que la salida real se utiliza en lugar de retroalimentar la producción estimada.

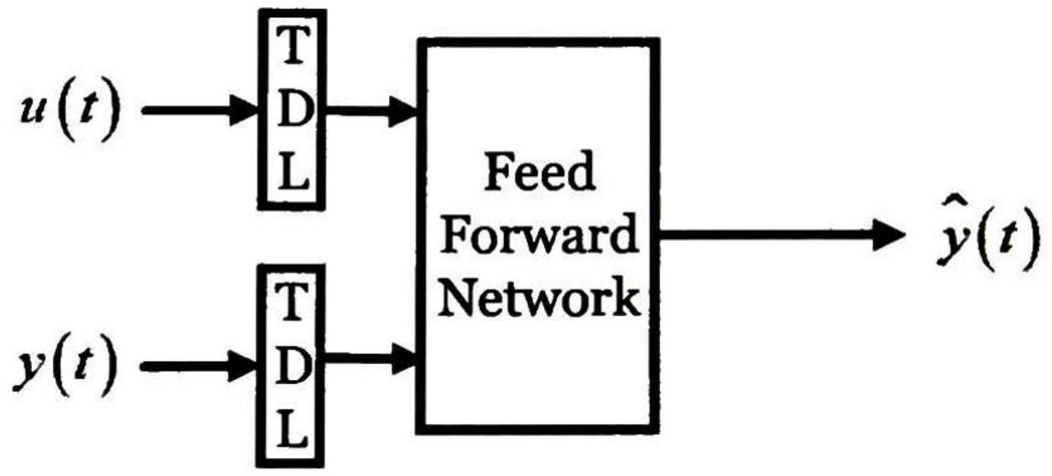


Fig. 4.4 Arquitectura Serie-Paralelo.

4.5 Formato punto fijo Vs Coma flotante

Unos de los factores importantes que se debe de tomar en cuenta para la implementación de algoritmos matemáticos de alta precisión son el formato numérico a emplear, ya que existen diversas formas de representar un número tanto en formato punto fijo y coma flotante.

Cabe mencionar que ambos métodos de representación presentan ventajas y desventajas, donde estas tienden a sobresalir según los requerimientos de las aplicaciones a implementar: como pueden ser velocidad de cálculo y precisión.

4.5.1 Representación en punto fijo

Las arquitecturas de punto fijo están basadas en una representación que contiene una cantidad fija de dígitos después del punto decimal. Al no requerir una unidad de coma flotante, la mayoría de los DSP de bajo costo utilizan esta arquitectura, aunque en determinados casos esta alternativa ofrece mejor rendimiento sobre todo cuando se requiere de velocidad de cálculo.

En una representación en punto fijo los *bits* a la izquierda del punto decimal se denominan *bits de magnitud* y representan los valores enteros, en cambio los *bits* a la derecha del punto decimal representan valores fraccionales (potencias inversas de 2), es decir el primer bit fraccional será $1/2$, el segundo $1/4$, el tercero $1/8$, y así sucesivamente hasta terminar con la cantidad de bits contenidos en la parte fraccionaria.

Para calcular la representación en punto fijo de un número dado se puede emplear la siguiente expresión:

$$2^{m-1} - \frac{1}{2^f} \tag{4.2}$$

donde m es el número de bits empleados para la parte entera, mientras que el valor de f representa la cantidad de bits en parte decimal.

4.5.1.1 Formato numérico $Q_{m,n}$

En un formato $Q_{m,n}$ es la forma más común de representar números decimales de punto fijo, se emplean m bits para representar en complemento 2 la parte entera y n bits para representar la parte fraccionaria en complemento 2. Para almacenar un número en formato $Q_{m,n}$ son necesarios de m+n+1 bits. El bit extra es usado para almacenar en la posición más significativa el signo del número. El rango entero representable es $(-2^m, 2^m-2^{-n})$ con una resolución de 2^{-n}.

Para el caso de un sistema digital de 16 bits con una representación en formato $Q_{4,12}$, indica que se emplean 3 enteros, 12 decimales y 1 signo.

Tabla 4.2 Distribución de bits en formato $Q_{m,n}$

S	*E*	*E*	*E*	*d*	*d*	*d*	*d*	*d*	*d*	*d*	*d*	*d*	*d*	*d*	*d*

La resolución del formato está dado por:

$$\frac{1}{2^n} \tag{4.3}$$

En el ejemplo anterior la resolución del dato será de 0.000244140625.

4.5.2 Representación en coma flotante IEEE 754

La representación en coma flotante IEEE 754[32] posee una gran ventaja con respecto a la representación en punto fijo: escalar las variables no es una preocupación y ha sido desarrollado para facilitar la portabilidad de los programas de un procesador a otro. Un número de coma flotante puede ser representado empleando precisión simple con 32 bits o precisión doble de 64 bits.

---

1. [32] IEEE Standard for Floating-Point Arithmetic (IEEE Std 754-2008), Revision of IEEE Std 754-4985. 29 August 2008.

Un dato en representación en compa flotante está compuesto por tres elementos importantes: signo, exponente y mantisa. El signo identifica si el número es positivo o negativo, el exponente contiene el valor de la potencia de la base, es decir la cantidad de bits que tiene que ser desplazada la magnitud del número para obtener el valor real, los espacios agregados son rellenados con ceros y por último la mantisa que contiene la magnitud del número en binario puro. Es común emplear los números en coma flotante en notación normalizada, donde se indica que el bit más significativo de la mantisa tiene que ser 1.

En la representación de datos en simple precisión, el exponente se encuentra codificado en un exceso en 127, es decir para un exponente menor a 127 la magnitud del número se encuentra por debajo de la unidad, en cambio si el exponente se encuentra por arriba de 127 la magnitud será mayor a la unidad.

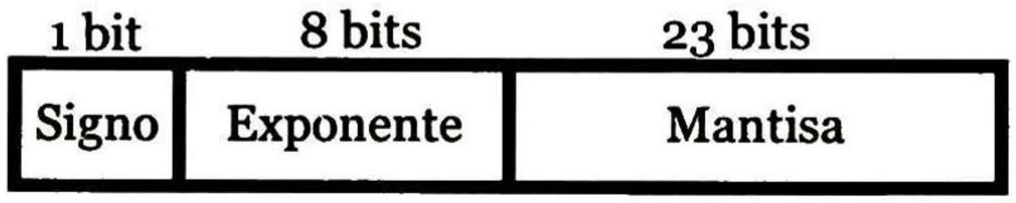


Fig. 4.5 Formato como flotante de simple precisión.

El rango numérico para la representación en simple precisión se encuentra entre $1.2x10^{-38}$... $3.4x10^{38}$.

Una representación en doble precisión en comparación con la precisión simple es diferencia en que la magnitud de los números puede ser representada con mayor exactitud debido a que la mantisa presenta mayor cantidad de bits, y el exponente permite un mayor rango de valores.

En está representación el exponente de dato presenta un exceso de 1023.

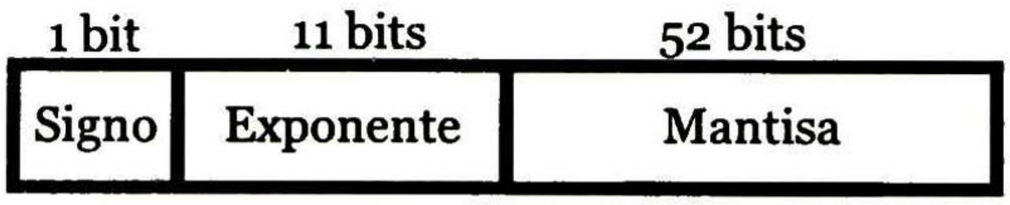


Fig. 4.6 Formato coma flotante de doble precisión.

4.5.2.1 Operaciones en coma flotante

Debido al tipo de notación que tiene una representación en coma flotante, las operaciones aritméticas no se pueden realizar de manera inmediata en comparación con la representación de punto fijo, por lo que para obtener los resultados de las operaciones aritméticas, es necesario emplear un algoritmo que permita calcular los componentes de la representación por separado.

4.5.2.1.1 Suma y resta en coma flotante

Para realizar una suma en coma flotante, es necesario seguir una serie de pasos para obtener el resultado correcto, el algoritmo necesario se describe a continuación:

- Extraer signos, exponentes y mantisas.
- Desplazar la mantisa del número con el exponente más pequeño a la derecha $|e_1-e_2|$ bits.
- Fijar el exponente del resultado a la máxima representación.
- Si la operación es suma y los signos son iguales, o si a operación es resta y los signos son diferentes, sumar las mantisas. En otro caso restarlas.
- Detectar overflow de la mantisa.
- Normalizar la mantisa, desplazándola a la derecha o a la izquierda hasta que el digito más significativo este delante del punto decimal.
- Redondear resultado y normalizar la mantisa si es necesario.
- Corregir el exponente en función de los desplazamientos realizados sobre la mantisa.
- Detectar overflow o underflow del exponente.

4.5.2.1.2 Multiplicación coma flotante

Realizar una multiplicación en coma flotante, de dos número es un proceso más sencillo en comparación a la suma. Ya que para este algoritmo, no es necesario ajustar los exponentes y reajustar las mantisas antes de realizar las operaciones necesarias.

Sean dos números dados:

$$\begin{aligned} X &= (-1)^{s1} * 1.mant1 * 2^{e1} \\ Y &= (-1)^{s2} * 1.mant2 * 2^{e2} \end{aligned} \tag{4.4}$$

donde $s1$ y $s2$ son lo valores del bit de signo sea cero para positivo o uno para negativo, $1.mantX$ es el valor de la mantisa normalizada y 2^{eX} es el exponente donde el 2 indica que el dato es binario y eX es la potencia a la que se eleva el exponente. La ecuación que rige la multiplicación en coma flotante está dada por (4.5).

$$Z = (-1)^{(s1 \oplus s2)} * (1.mant1 * 1mant2) * 2^{(e1+e2-sesgo)} \qquad (4.5)$$

El procedimiento para una multiplicación en coma flotante se muestra a continuación.

- El signo del resultado es igual a la Or-Exclusiva de los dos signos de los operandos.
- La mantisa del resultado es igual al producto de las mantisas. Este producto es sin signo.
  Dado que los operandos están comprendidos entre 1 y 2 el resultado será:

$$1 < r < 4 \qquad (4.6)$$

  en este paso se puede requerir una normalización, mediante desplazamientos a la derecha y ajuste del exponente del resultado.
- Si la mantisa resultante es del mismo tamaño que la de los operandos habrá que redondear. El redondeo puede implicar la necesidad de normalizar.
- El exponente del resultado es igual a la suma de los exponentes de los operandos. Considerando que usamos una representación sesgada del exponente, al hacer esta suma se están sumando dos veces el sesgo, y por lo tanto habrá que restar este sesgo una vez para obtener el resultado correcto.

4.5.2.1.3 División en coma flotante

El proceso de división es algo similar al de multiplicación, donde el cálculo de los signos se realiza con una Or-Exclusiva, la mantisa se calcula realizando una división y en este caso los exponentes se restan. Dados los números de (4.4), la ecuación que rige la división en coma flotante está dada por (4.7) y el algoritmo para su implementación se muestra a continuación.

$$Z = (-1)^{(s1 \oplus s2)} * (1.mant1 / 1mant2) * 2^{(e1-e2)} \qquad (4.7)$$

- Al hacer la resta de los exponentes hay que considerar que los sesgos se anularán y por tanto al resultado hay que sumarle el sesgo.

- Al trabajar con números normalizados, la mantisa del resultado será:

$$0.5 < r < 2 \tag{4.8}$$

lo que implica que se tendrá que realizar una normalización mediante un desplazamiento de bits hacia la izquierda.

4.6 Diseño de neurona artificial

Como se mencionó en la sección 3.2 la neurona artificial está constituida por entradas, pesos, polarizaciones, unidad de procesamiento y función de activación. El diseño propuesto para la neurona artificial, puede ser segmentado en bloques los cuales contienen las operaciones necesarias para el funcionamiento de una neurona simple de múltiples entradas, las funciones matemáticas realizadas por la neurona son implementadas en formato coma flotante de precisión simple con la finalidad de anexarle mayor exactitud a los resultados y tener un punto de comparación contra la misma arquitectura realizada en software.

En el diagrama de la Fig. 4.7 se muestra la arquitectura de una neurona artificial, y los bloques en los que ha sido segmentado, estos se explican a más detalle en una sección posterior.

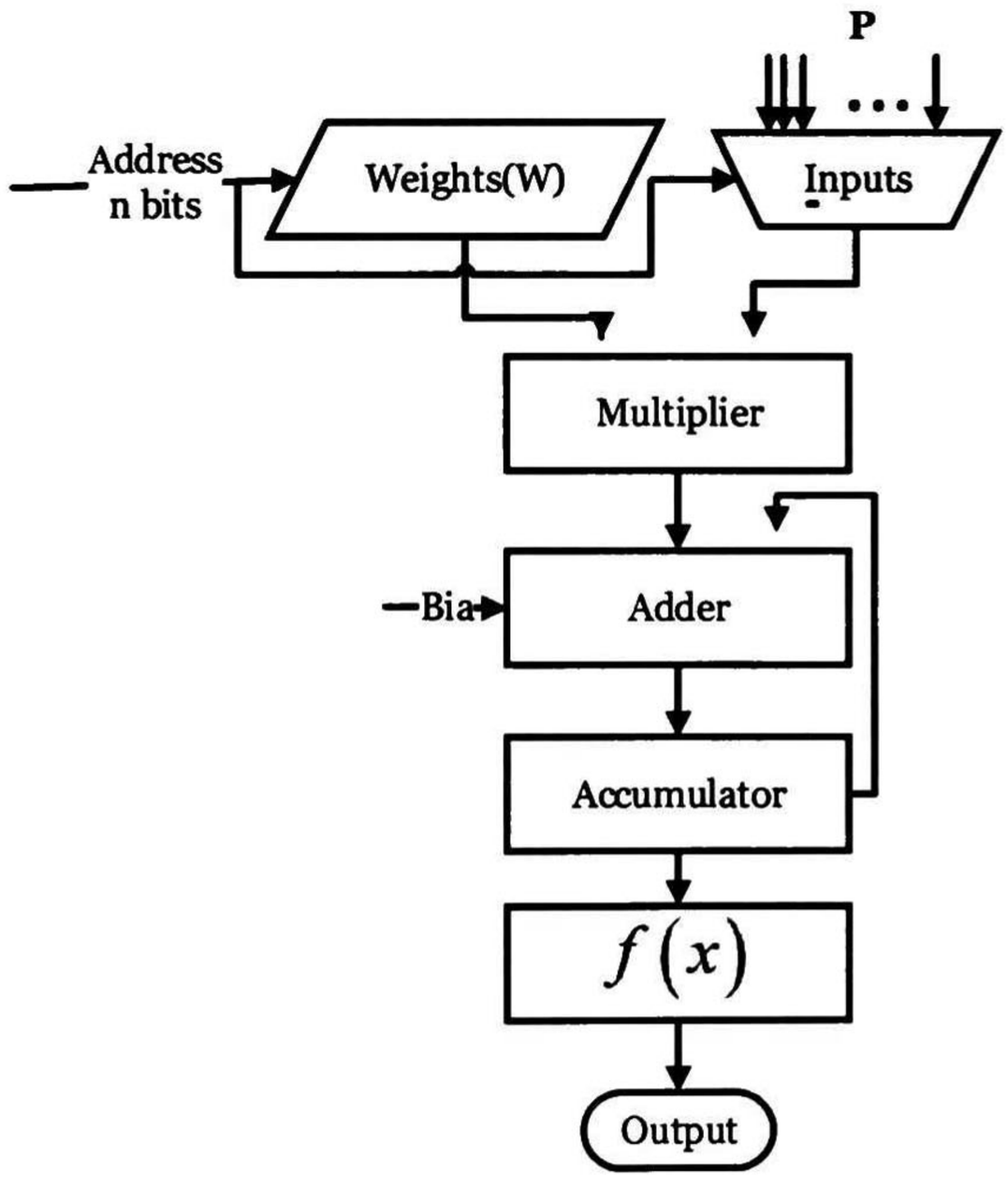


Fig. 4.7 Arquitectura de una neurona artificial en FPGA.

Remitiéndome a publicaciones sobre implementación de neuronas artificiales en FPGA, existe una publicación con un diseño similar donde la arquitectura toma una muestra de una solución química mediante un sensor, la medición analógica es convertida al dominio digital mediante un ADC y con un conjunto de neuronas artificiales programadas en el dispositivo reconfigurable, se determina la concentración de la solución[33]. A diferencia del diseño que se propone, esta arquitectura fue implementada en punto fijo en formato $Q_{n,m}$ y complemento a 2 para poder representar valores positivos y negativos.

La neurona propuesta para el diseño, está constituida con una memoria RAM de primero lectura que permite almacenar la matriz de pesos (W), un multiplexor que selecciona la entrada, un multiplicador, un sumador, un acumulador en coma flotante y por ultimo una función de activación.

4.6.1 Memoria RAM

La memoria de acceso aleatorio (*Random Access Memory*) es empleada comúnmente como memoria de trabajo y se denomina de acceso aleatorio porque se puede leer o escribir en una posición de memoria con un tiempo de espera igual que en cualquier posición, no siendo necesario un orden para acceder a la información.

Los dispositivos FPGA de Xilinx proveen dos tipos de memoria RAM que son distribuidas y de bloque.

La RAM distribuida configura a las LUTs contenidas en un CLB y la RAM de bloque emplea los módulos dedicados incorporados en el dispositivo.

Existen tres maneras de incorporar una RAM en un diseño:

- Empleando inferencia automática, donde la herramienta de síntesis de Xilinx, se encarga de configurar y emplazar la RAM declarada.

- Empleando *CORE Generator*[34], que es una herramienta de Xilinx altamente parametrizable que provee de bloques de propiedad intelectual (IP) como son unidades de almacenamiento o bloques dedicados al procesamiento de señales.

- Instanciando elementos dedicados de una librería *UNISIM* o *UNIMACRO*.

---

[33] Mohamad, Khairudin, Mohamad Faiz Omar Mahmud, Fadzilatul Husna Adnan, and Wan Fazlida Hanim Abdullah. "Design of single neuron on FPGA." In Humanities, Science and Engineering Research (SHUSER), 2012 IEEE Symposium on, pp. 133-136. IEEE, 2012.

[34] http://www.xilinx.com/tools/coregen.htm

Los bloques RAM de Xilinx son bloques Dual-Port, donde cada puerto es totalmente independiente y puede ser configurado con diferentes anchos y profundidad. Algunos tipos de memorias RAM son:

Primero lectura: Son aquellas que durante la escritura muestran en las terminales de salida la información que estaba guardada en la posición que está siendo direccionada.

Primero escritura: Son aquellas que durante el proceso de escritura muestran en las terminales la información que se está almacenando.

No cambio: No proporcionan ningún dato en la salida mientras se está realizando la operación de escritura.

Para almacenar la matriz de pesos requerida por la neurona artificial, se ha implementado una memoria RAM de primero lectura con la herramienta *CORE Generator*, con un ancho de palabra de 32 bits necesarios para la representación de un dato en coma flotante de precisión simple y la profundidad varia con respecto al número de entradas y retardos de la neurona.

Para agregar un bloque con la herramienta *CORE Generator* en la ventana *"Design"*, se anexa una nueva fuente al proyecto esta opción se muestra en Fig. 4.8.

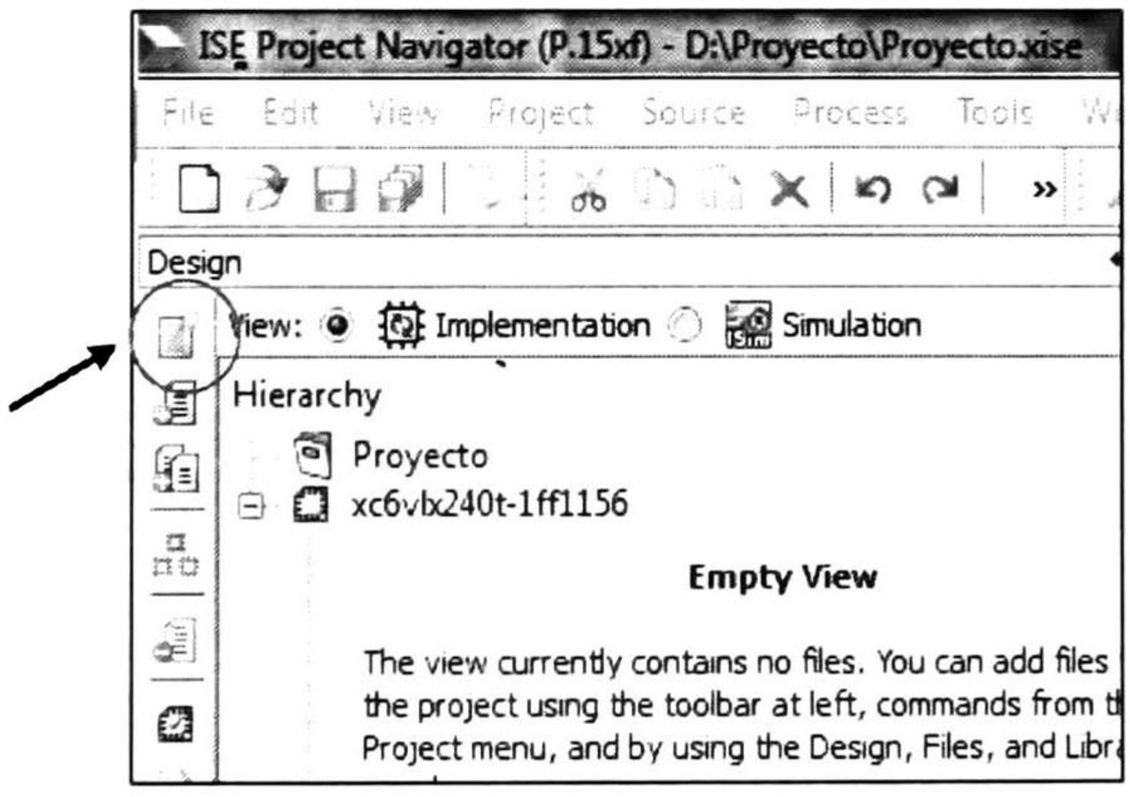


Fig. 4.8 Insertar nueva fuente en ISE Project Navigator.

Posteriormente seleccionar el tipo de fuente que se muestra en Fig. 4.9, que será *"IP (CORE Generator & Architecture Wizard)"*, asignar nombre a la fuente presionar el botón de "Next".

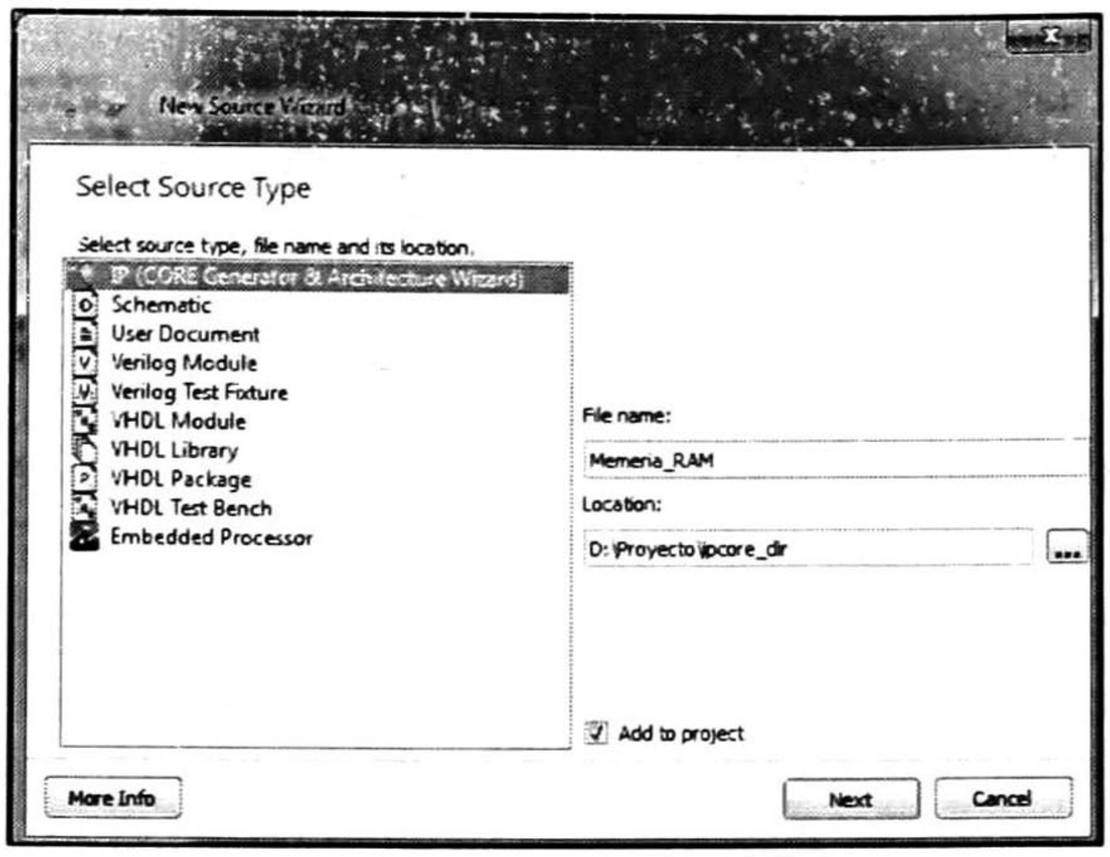


Fig. 4.9 Selección de fuente IP (CORE Generator).

A continuación buscar el bloque requerido en el árbol de componentes, en este caso seleccionar "*Block Memory Generator*", para generar una memoria RAM de bloque, presionar "*Next*" hasta finalizar esta opción se muestra en Fig. 4.10.

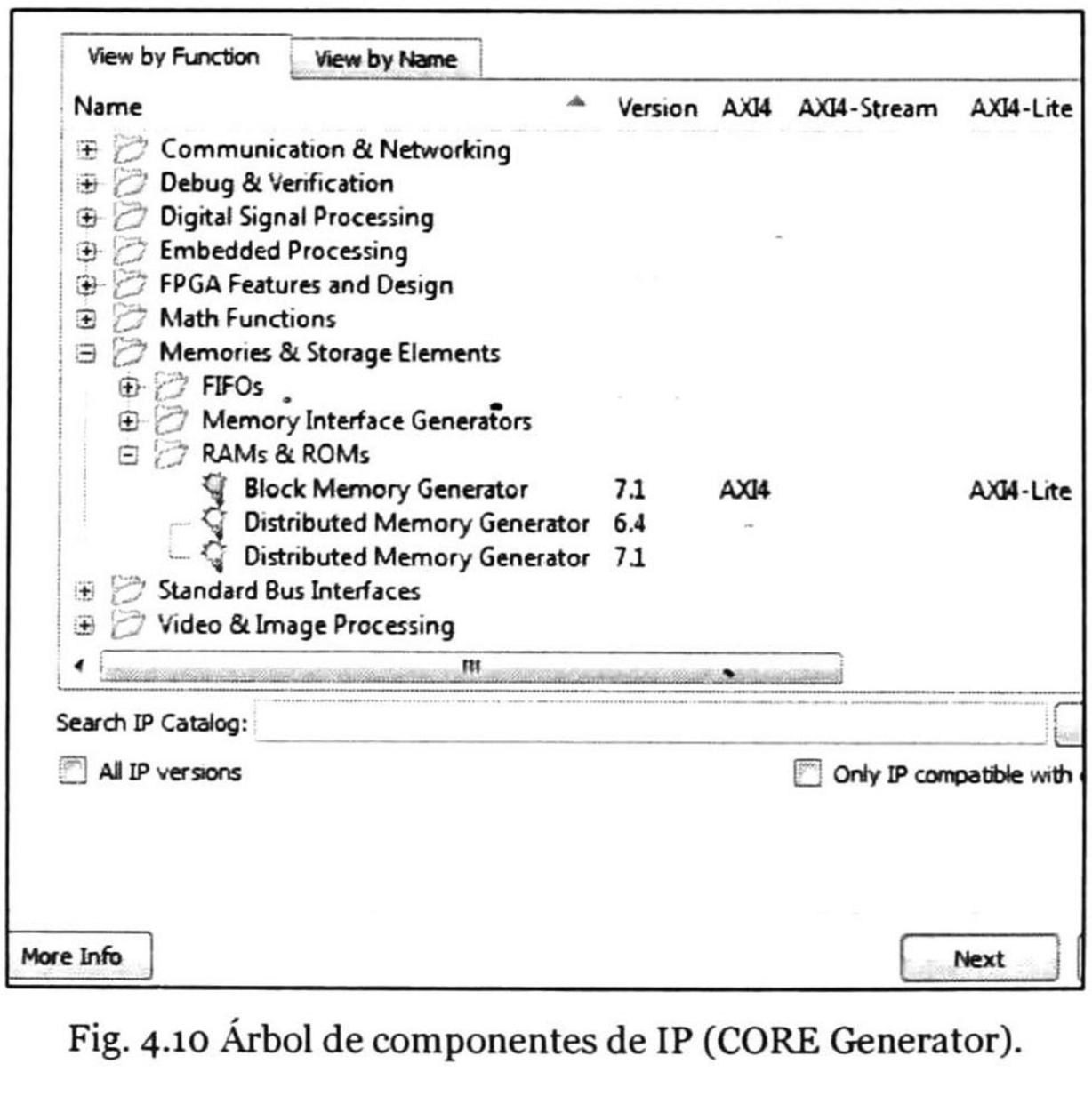


Fig. 4.10 Árbol de componentes de IP (CORE Generator).

La siguiente ventana contiene un menú de características propias del bloque seleccionado, para una memoria que se muestra en Fig. 4.11 es importante definir si será de puerto simple o doble, el modo de operación, ancho de palabra, profundidad de la memoria y si es necesario inicializarla con valores propios del usuario. Todas estas características se encuentran distribuidas en un conjunto de 6 ventanas de configuración.

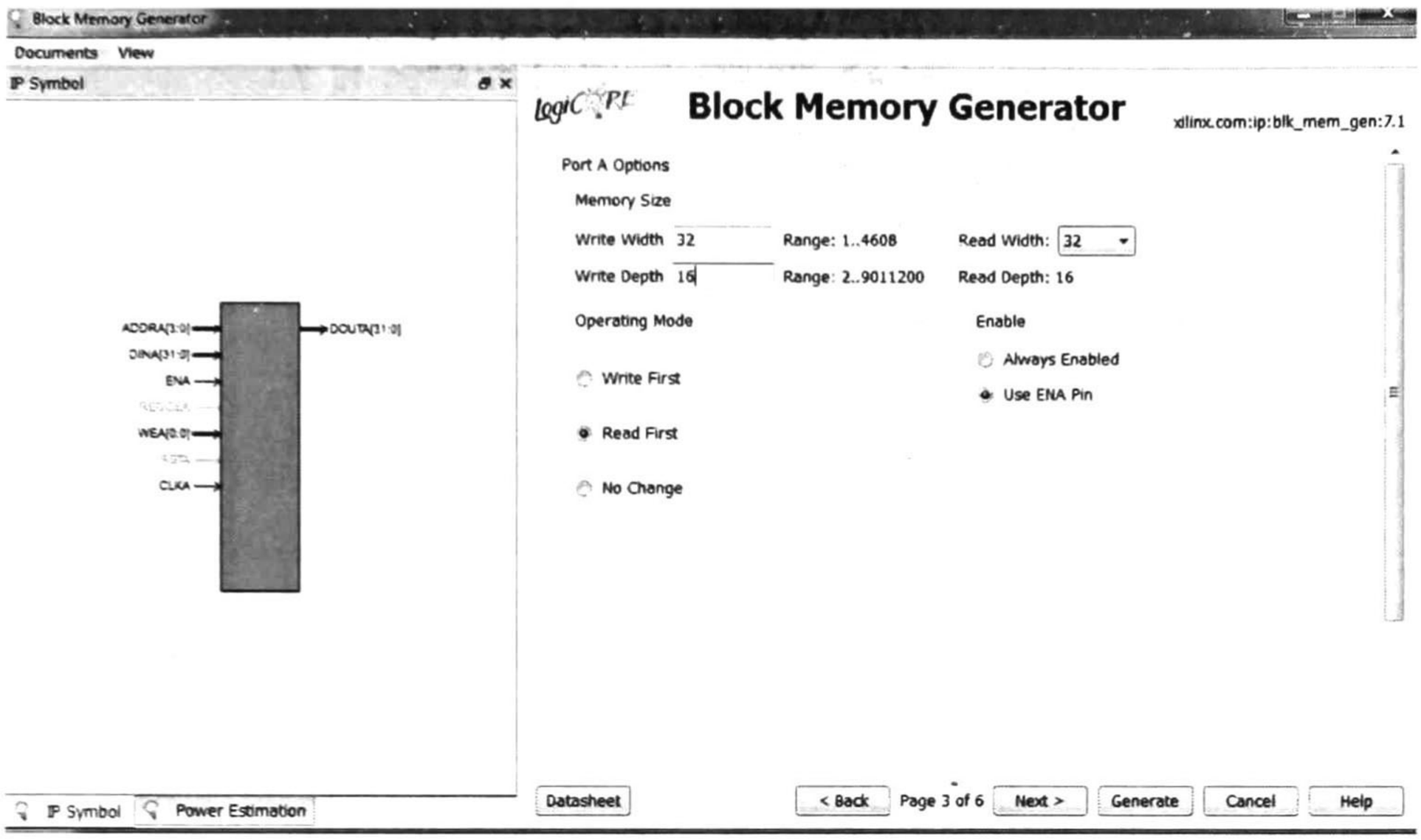


Fig. 4.11 Configuración de una memoria RAM de primero lectura.

Si es necesario inicializar la memoria RAM, se puede elaborar un archivo con extensión .coe" del inglés "*CORE Generator Coefficients*", este archivo se puede elaborar en un block de notas y cambiar la extensión ".txt" a ".coe".

Para que el archivo de inicialización sea válido y reconocido por la herramienta de Xilinx, se debe poner como línea de cabecera la raíz del dato ya sea hexadecimal, decimal, binario y octal, posteriormente la línea de inicialización de los vectores, y a continuación los datos separados cada uno de ellos por una "," y finalizando con ";".

```
memory_initialization_radix=16;
memory_initialization_vector=
    3e8b73c0,
    beb05dfc,
    be3cd827,
    3e978478,
    bf099bd3,
    3f425fd0,
    bf0698a0,
    3f4efb57,
    3d19c0e1,
    3e672d19,
    bdb74d47,
    be1b4318,
    be67dae8,
    bf047e54,
    bea4244c,
    bf33d27b;
```

Fig. 4.12 Archivo de inicialización de memoria RAM.

Una vez que se ha cargado el archivo es posible visualizar los datos que inicializarán la memoria, si no se requieren más configuraciones se puede finalizar la ventana de configuración presionando el botón de "Next" hasta finalizar.

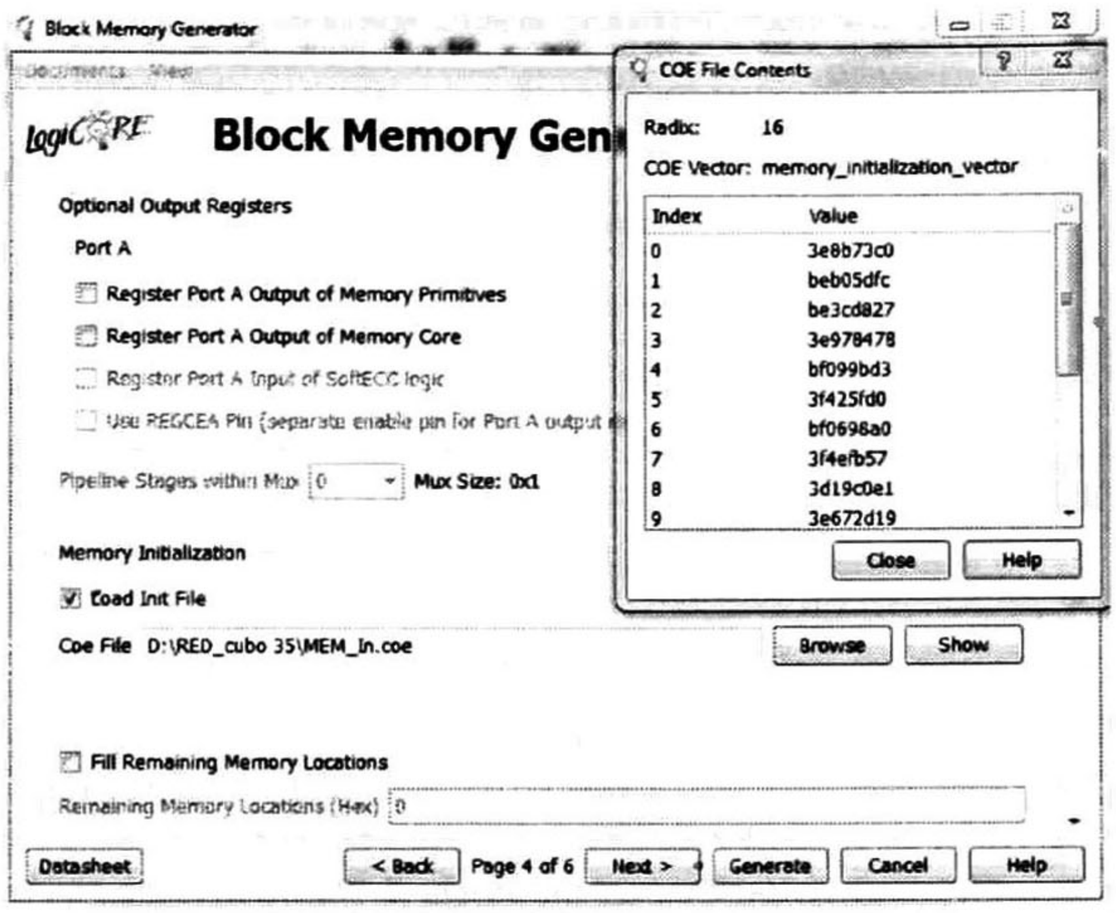


Fig. 4.13 Insertar archivo de inicialización de memoria.

Una vez generado el bloque puede ser usado instanciando en el archivo principal.

4.6.2 Multiplexor

El multiplexor empleado en la neurona se utiliza para seleccionar la entrada correspondiente al peso que se pretende multiplicar. Ambos datos son seleccionados con el mismo bus de direcciones que se ha conectado en la memoria, de esta manera se pueden evitar errores al direccionar las datos de forma incorrecta. El modulo se declaró en lenguaje Verilog donde, la entrada es de ancho de palabra de 32 bits.

4.6.3 Multiplicador en coma flotante

El bloque multiplicador es el equivalente a la operación realizada por las conexiones ponderadas cuando se les presenta una entrada en una neurona artificial, este bloque es capaz de realizar una operación de dos números con un mínimo de 5 ciclos de reloj y es capaz de trabajar con datos normalizados es decir donde el bit más significativo de la mantisa es uno. La facilidad que ofrece el formato para representar un gran rango de valores con la misma cantidad de bit, es una característica que permite realizar multiplicaciones consecutivas sin provocar desbordamiento del resultado. En el diagrama que se muestra en Fig. 4.14 se describe el proceso de multiplicación, donde se puede denotar que la cantidad de

ciclos de reloj empleados para obtener un resultado consistente, puede variar con la cantidad de bits que se tengan que normalizar del resultado. La operación de multiplicación solo se puede realizar cuando se han colocado los dos operando y posteriormente se activa la señal de "start", el proceso multiplicación iniciara y una vez que el resultado corrector este presente, se activará un bit de finalizado.

Existen medios alternativos de implementar las multiplicaciones en coma flotante, y una de ellas es empleando un *CORE* de Xilinx especializado en funciones aritméticas en este formato, en el menú de configuración se puede seleccionar si se quiere que se implemente empleado los bloques DSP del FPGA o solamente bloques lógicos.

Las multiplicaciones en coma flotante no solo aplican para valores numéricos en binario, por lo que se han propuesto arquitecturas que pueden resolver multiplicaciones en coma flotante decimal[35].

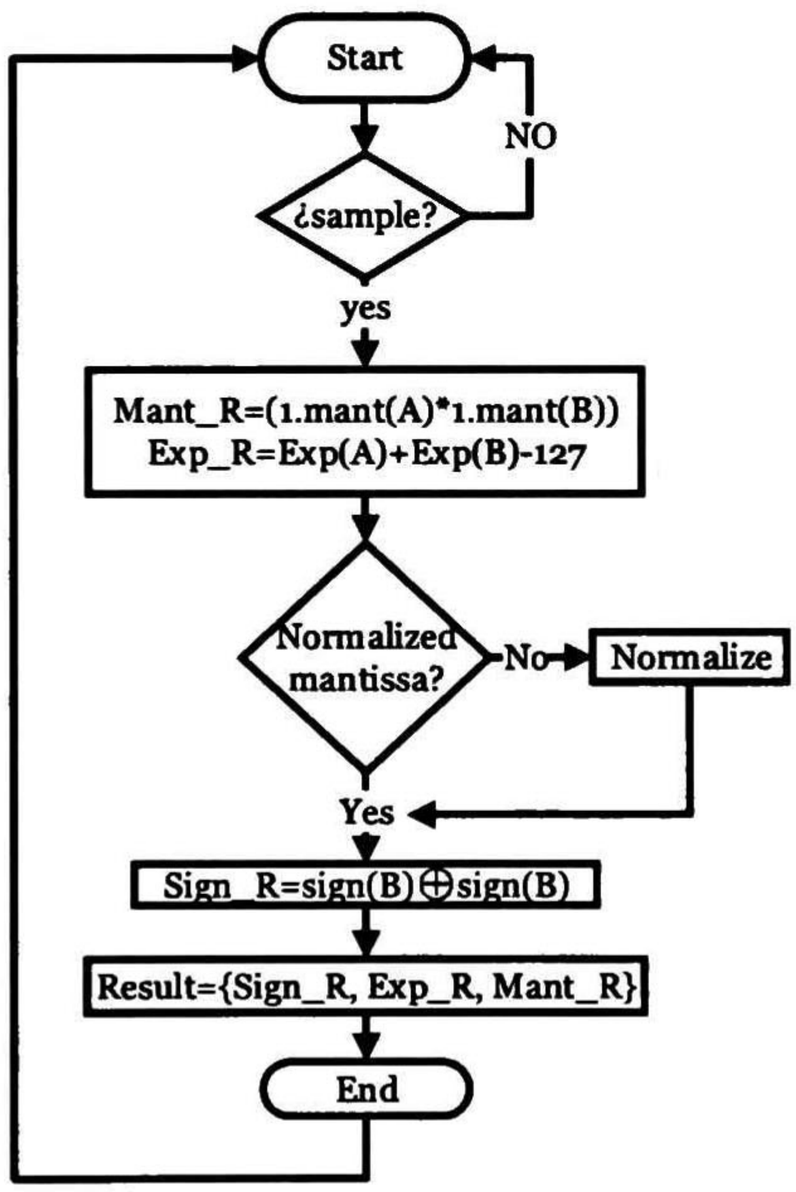


Fig. 4.14 Algoritmo para multiplicación en coma flotante.

4.6.4 Sumador y acumulador en coma flotante

El sumador y acumular en conjunto funcionan como la unidad de procesamiento de la neurona artificial, y esta se encarga de realizar la sumatoria de todas las conexiones ponderadas conectadas a la neurona.

---

[35] Minchola, Carlos, and Gustavo Sutter. "A FPGA IEEE-754-2008 Decimal64 Floating-Point Multiplier." In Reconfigurable Computing and FPGAs, 2009. ReConFig'09. International Conference on, pp. 59-64. IEEE, 2009.

Este bloque solo puede realizar la suma de dos datos a la vez, por lo que es necesario procesar la suma de forma secuencial ingresando un dato a la vez. Para ellos se requiere de algún tipo de control para el flujo de datos.

La operación de suma fue implementada con un *CORE* de Xilinx para operaciones en coma flotante, que permite realizar sumas y restas sin necesidad de cambiar configuraciones del bloque. Entre las configuraciones importantes está la selección de la operación a realizar y precisión del dato, aunque es importante agregar en las configuraciones las señales de control para el bloque. Estas señales son "*operation_ND*" que es la señal de inicio y "*RDY*" que es la señal de operación finalizada.

El acumulador es un bloque que almacena los resultados y actualiza cada vez que se realiza una nueva suma de datos.

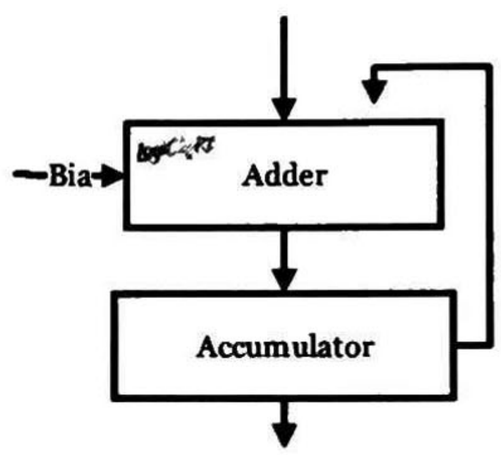


Fig. 4.15 Sumatoria en coma flotante.

4.6.5 Función de activación

La función de activación, es la que define el nivel de excitación de una neurona, y para el modelado no lineal es necesario la implementación de una función que sea no lineal. Para ellos existen diversas funciones posibles a emplear, pero en el caso de una red neuronal NARX, la función correspondiente en la capa oculta es una tangente hiperbólica.

La tangente hiperbólica comprime los valores de entrada en un rango de datos positivos y negativos con valor de ±1, y debido a la complejidad de esta función hiperbólica, realizar la implementación de forma directa en lenguaje de descripción de hardware resulta ser una tarea difícil. Esta función es de gran importancia para la implementación de redes neuronales, motivo por el cual los investigadores han buscado alternativas de su implementación en dispositivos reconfigurables.

Existen diferentes métodos de implementación como: algoritmo de CORDIC[36], uso de LUTs, métodos híbridos y por segmentos.

El algoritmo de CORDIC (*Coordinate Rotation Digital Computer*) o computadora digital para rotación de coordenadas fue propuesto para

---

[36] Qian, Meng. "Application of CORDIC algorithm to neural networks VLSI design. In Computational Engineering in Systems Applications, IMACS Multiconference on, vol. 1, pp. 504-508. IEEE, 2006.

calcular funciones trigonométricas mediante rotación de vectores. Este método iterativo rota vectores en ciertos ángulos determinados y se basa exclusivamente en sumas y desplazamientos. El algoritmo original describe, la rotación de un vector bidimensional en el plano cartesiano, su funcionamiento se basa en la formula general para rotación de vectores.

$$\begin{aligned} x' &= x\cos\theta - y\sin\theta \\ y' &= y\cos\theta - x\sin\theta \end{aligned} \tag{4.9}$$

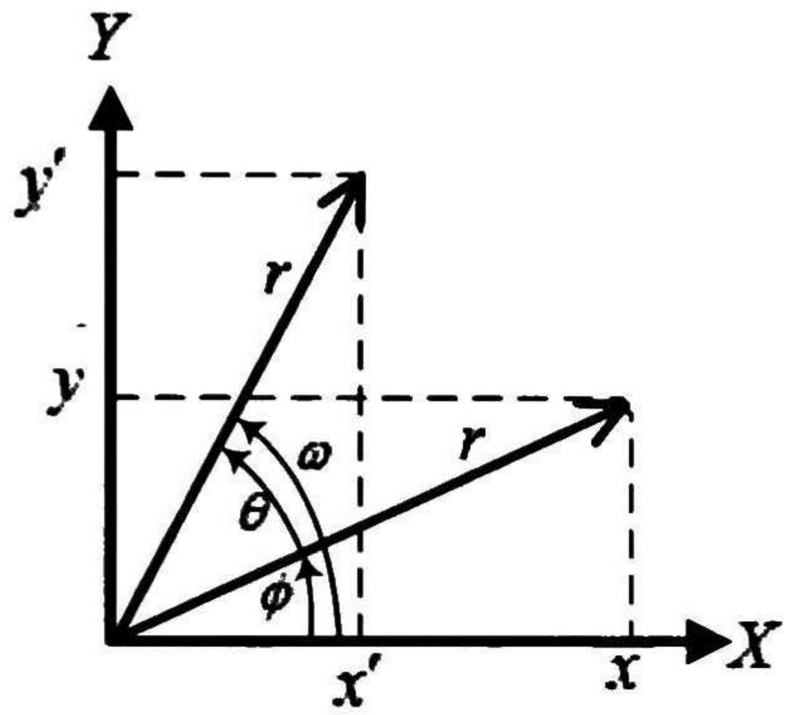


Fig. 4.16 Grafica que describe la rotación de vectores.

El algoritmo de CORDIC opera normalmente en dos modos, el primero se denomina rotación, el algoritmo rota el vector de entrada un ángulo específico que se introduce como parámetro. El segundo modo, denominado vectorización, rota el vector de entrada hacia el eje X, acumulando el ángulo necesario para efectuar el movimiento. Este algoritmo tiene la capacidad de calcular las funciones seno, coseno y arco tangente en sus formas circulares e hiperbólicas.

La herramienta *CORE Generator*, proporciona un bloque en su librería capaz de resolver funciones trigonométricas e hiperbólicas que se han mencionado anteriormente empleando el método de algoritmo de CORDIC, entre las características principales que destacan es que emplea un formato punto fijo $Q_{2,n}$ y complemento a 2 para representar valores positivos y negativos.

Debido a que no se puede calcular de forma directa la tangente hiperbólica de un dato, se tiene que obtener el resultado de forma inferida por medio de una división que se muestra en la ecuación (4.10).

$$\tanh(x) = \frac{\text{senh(x)}}{\cosh(x)} \tag{4.10}$$

Este medio de cálculo carece de eficiencia, ya que como la arquitectura que se ha planteado está diseñada para trabajar en formato coma flotante y se tienen que realizar conversiones de formato para poder obtener el resultado de la función de activación. La desventaja de mayor impacto en este medio de solución, es que el bloque proporcionado por Xilinx, no permite realizar el cálculo de un función hiperbólica con un entrada mayor a ±1 radian, motivo por el cual se desechó esta propuesta. El algoritmo que se había planteado se muestra a en el diagrama de la Fig. 4.17.

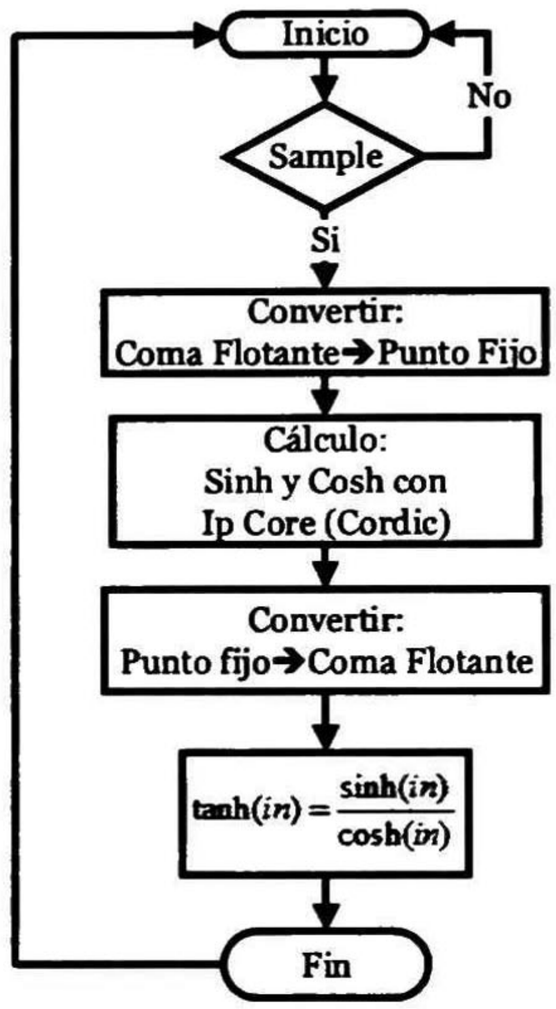


Fig. 4.17 Algoritmo para cálculo de tangente hiperbólica por CORDIC.

El uso de LUTs[37] (*Look-Up Table*) es otro medio de solución para la implementación de funciones trigonométricas y es la programación de una tabla donde para cada valor de entrada se tiene el correspondiente valor de salida. Este no es el método más óptimo pero ha sido muy útil en la implementación de funciones complejas donde resulta difícil el algoritmo a programar. Las LUTs solo tienen un número finito de valores correspondientes a entradas y salidas por lo que el número de datos es limitado así como también el ancho de palabra está definido por el usuario lo que incorpora falta de precisión en los resultados. El uso de un tabla conlleva a la implementación de un algoritmo de búsqueda, que es el encargado de encontrar la dirección de memoria donde se encuentra almacenado el valor de la salida, entre más eficiente sea el algoritmo de búsqueda los datos pueden ser presentados con mayor velocidad. Esta implementación con respecto a la anterior se ejecuta en menos ciclos de

[37] Leboeuf, Karl, Roberto Muscedere, and Majid Ahmadi. "Performance analysis of table-based approximations of the hyperbolic tangent activation function." In Circuits and Systems (MWSCAS), 2011 IEEE 54th International Midwest Symposium on, pp. 1-4. IEEE, 2011.

reloj, ya que solo se calcula la dirección y se localiza el resultado en la tabla.

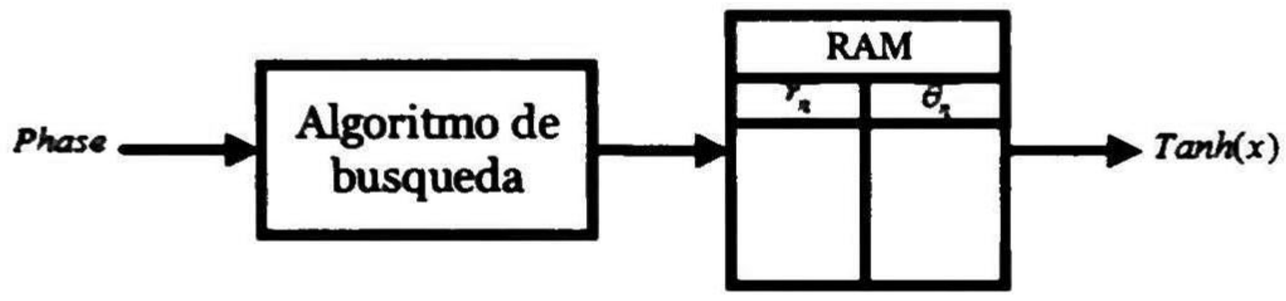


Fig. 4.18 Diagrama básico de una LUT.

La implementación por segmentos, consiste en fraccionar la función hiperbólica en segmentos lineales[38] para intentar recrear la función lo más parecido posible. Este método presenta un grado de error mayor en comparación de los mencionados con anterioridad, debido a que la precisión es afectada directamente con la cantidad de segmentos a fraccionar la función y los segmentos no siempre se ajustan con mucha exactitud.

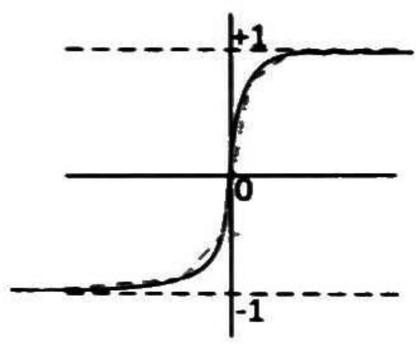


Fig. 4.19 Aproximacion por segmentos.

Algunos algoritmos híbridos[39] combinan dos a más métodos para obtener un mejor rendimiento en los resultados. *RALUT*[40] (*Range Addressable Look-Up Table*) es un método que emplea segmentos lineales para aproximar la tangente hiperbólica y para conseguir mayor precisión se almacenan valores de ajuste que son sustraídos de la función calculada con anterioridad, la sustracción puede ser implementada con simple circuito combinacional.

---

[38] Ferreira, Pedro, Pedro Ribeiro, Ana Antunes, and Fernando Morgado Dias. "A high bit resolution FPGA implementation of a FNN with a new algorithm for the activation function" Neurocomputing 71, no. 1 (2007): 71-77.

[39] Sartin, Maicon A., and Alexandre CR da Silva. "Approximation of hyperbolic tangent activation function using hybrid methods." In Reconfigurable and Communication-Centric Systems-on-Chip (ReCoSoC), 2013 8th International Workshop on, pp. 1-6. IEEE, 2013.

[40]Namin, Ashkan Hosseinzadeh, Karl Leboeuf, Huapeng Wu, and Majid Ahmadi. "Artificial neural networks activation function HDL coder." In Electro/Information Technology, 2009. eit'09. IEEE International Conference on, pp. 389-392. IEEE, 2009.

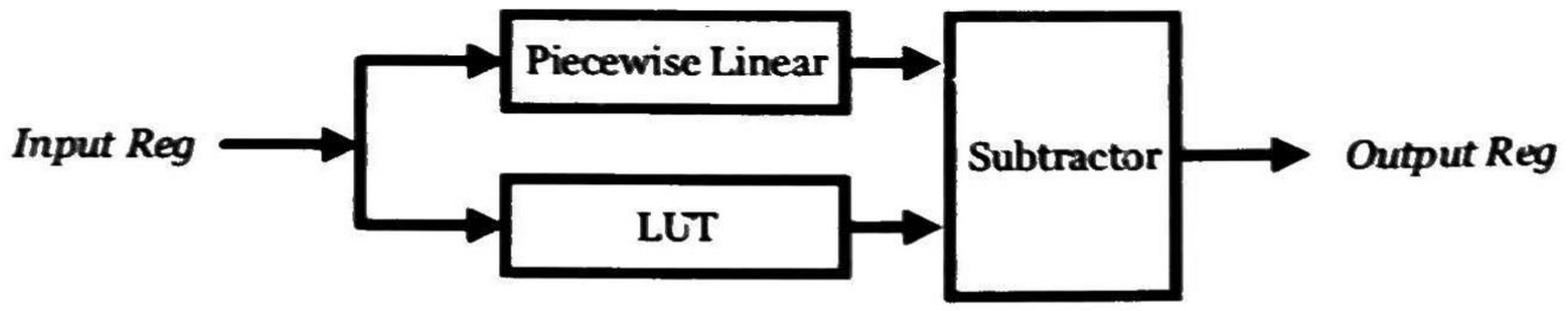


Fig. 4.20 Implementación de un método hibrido.

El método propuesto para la implementación de la función de activación es el uso de series de Taylor, combinado con segmentación.

La función matemática que describe a una tangente hiperbólica como se ha mostrado en (4.10), puede ser simplificada en funciones exponenciales.

$$\tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}} \tag{4.11}$$

La programación de una función exponencial es de gran dificultad en los dispositivos reconfigurables, por lo que es necesario la implementación de una aproximación de estas funciones, es donde juegan un papel importante las series de Taylor.

Por definición, la función exponencial puede ser representada en serie que se muestra en (4.12).

$$e^{x} = 1 + x + \frac{x^{2}}{2!} + \frac{x^{3}}{3!} + \frac{x^{4}}{4!} + \frac{x^{5}}{5!} + \ldots \frac{x^{n}}{n!} \tag{4.12}$$

Esta función es válida para valores de x positivos y negativos. Antes de realizar la sustitución de (4.12) en (4.11), es importante determinar el grado del polinomio a emplear. Ya que de este depende la precisión que le incorporar a la función final. Realizando un análisis de la función en Matlab y calculando el error de las funciones resultados, se llega a la conclusión que con un polinomio de grado 9, la función se ajusta perfectamente hasta con una entrada de ±3.33 radianes. El desarrollo de la función final de la implementación de tangente hiperbólica se muestra a continuación:

$$\begin{aligned} e^{x} - e^{-x} &= 2\left(x + \frac{x^{3}}{3!} + \frac{x^{5}}{5!} + \frac{x^{7}}{7!} + \frac{x^{9}}{9!} \right) \\ e^{x} + e^{-x} &= 2\left(1 + \frac{x^{2}}{2!} + \frac{x^{4}}{4!} + \frac{x^{6}}{6!} + \frac{x^{8}}{8!} \right) \end{aligned} \tag{4.13}$$

los equivalentes de los exponenciales, son sustituidos en la función de la tangente hiperbólica.

$$\tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}} = \frac{2\left(x + \frac{x^3}{3!} + \frac{x^5}{5!} + \frac{x^7}{7!} + \frac{x^9}{9!}\right)}{2\left(1 + \frac{x^2}{2!} + \frac{x^4}{4!} + \frac{x^6}{6!} + \frac{x^8}{8!}\right)} \qquad (4.14)$$

Reduciendo algebraicamente la función se obtiene la ecuación equivalente que se implementará en el FPGA.

$$\tanh(x) = \frac{362880x + 60480x^3 + 3024x^5 + 72x^7 + x^9}{362880 + 181440x^2 + 15120x^4 + 504x^6 + 9x^8} \qquad (4.15)$$

La ecuación final, se desglosa en funciones más simples, se puede implementar empleado multiplicaciones, suma y división en coma flotante.

El sistema empleado se considera hibrido, debido a que cuándo se requiere calcular tangente hiperbólica un valor cuya entrada se encuentra en el rango de ±3.33 radianes, el algoritmo aplica el polinomio (4.15), para valores mayores a estos el resultado es aproximado a ±1 según sea el caso.

En la Fig. 4.22 se muestra la función segmentada implementada, el segmento 1 y 3 son lineales, mientras que el segmento 2 es no lineal correspondiente al polinomio.

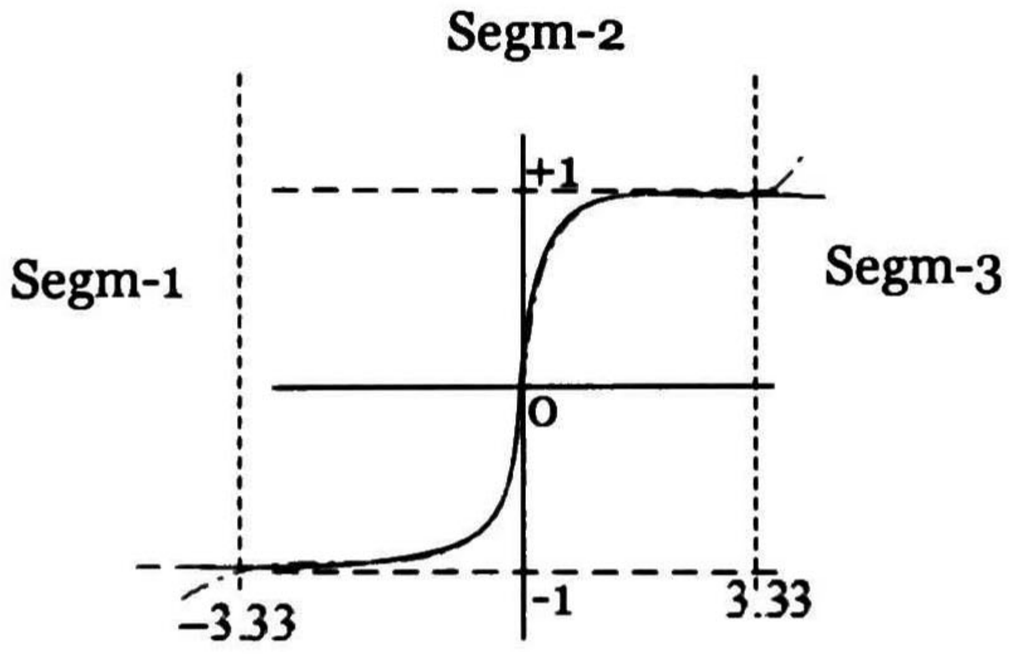


Fig. 4.21 Tangente hiperbólica implementada.

El diagrama de a bloques del diseño implementado se muestra a continuación.

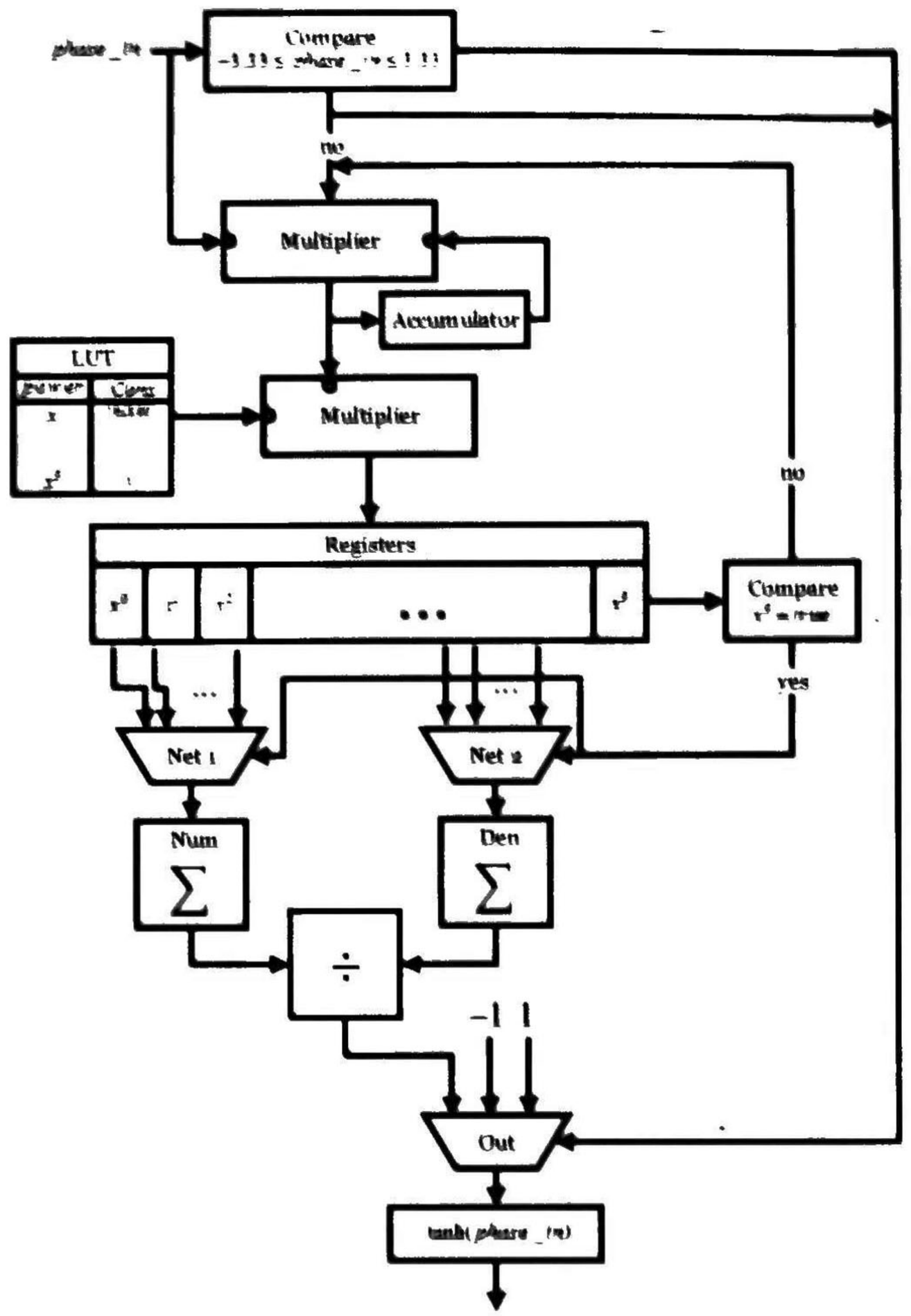


Fig. 4.22 Diagrama a bloques de la tangente hiperbólica.

En la implementación de la arquitectura mostrada en Fig. 4.22, se hizo uso de un CORE de Xilinx para realizar la división en coma flotante. Para el control de la arquitectura se empleó una máquina de estados que permite controlar correctamente el flujo de señales.

Una vez finalizada la arquitectura los datos obtenidos son simulados y se obtienen los errores absolutos y relativos en Matlab. La función implementada alcanza el error máximo cuando la entrada se encuentra en la intersección de los segmentos el error obtenido es menor al 1%.

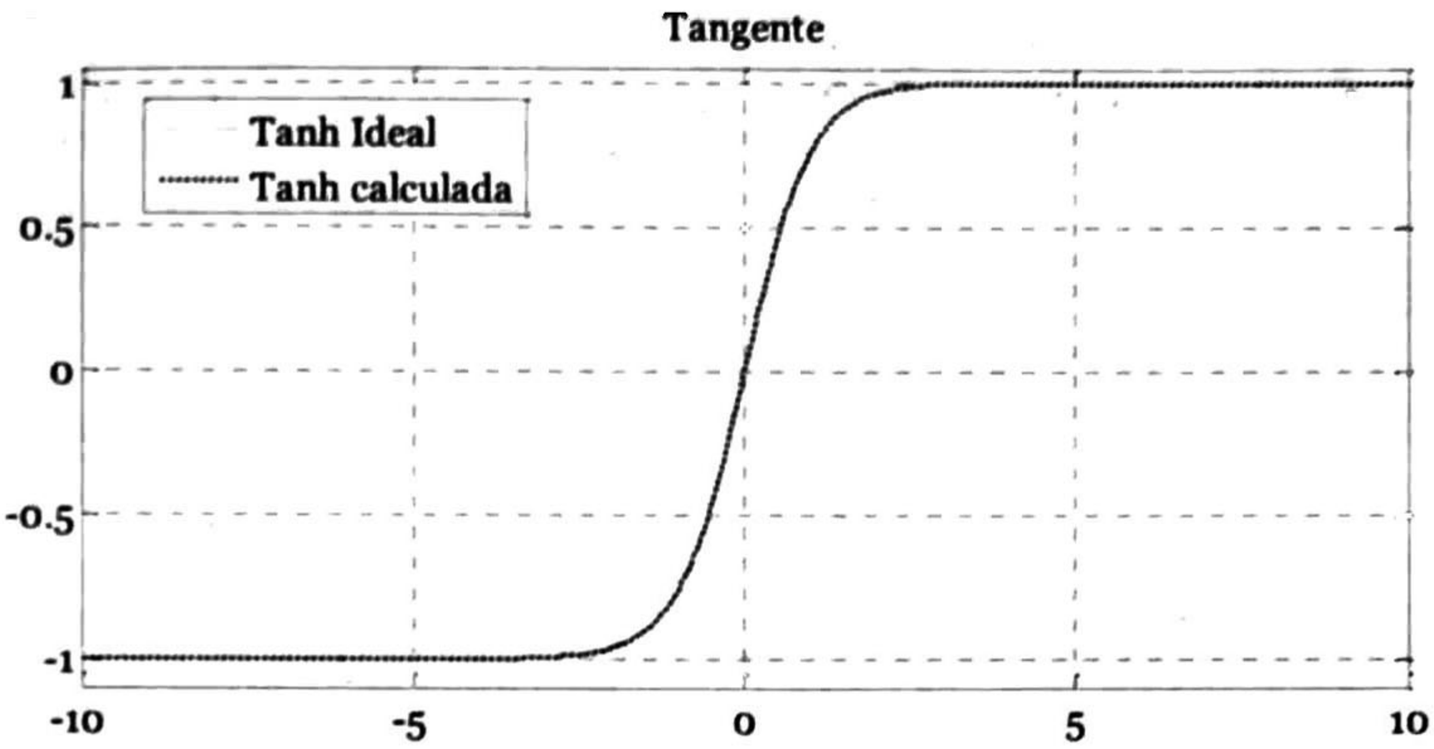


Fig. 4.23 Simulación de tangente hiperbólica ideal y calculada.

La línea puenteada muestra la gran correspondía del algoritmo implementado con la salida ideal.

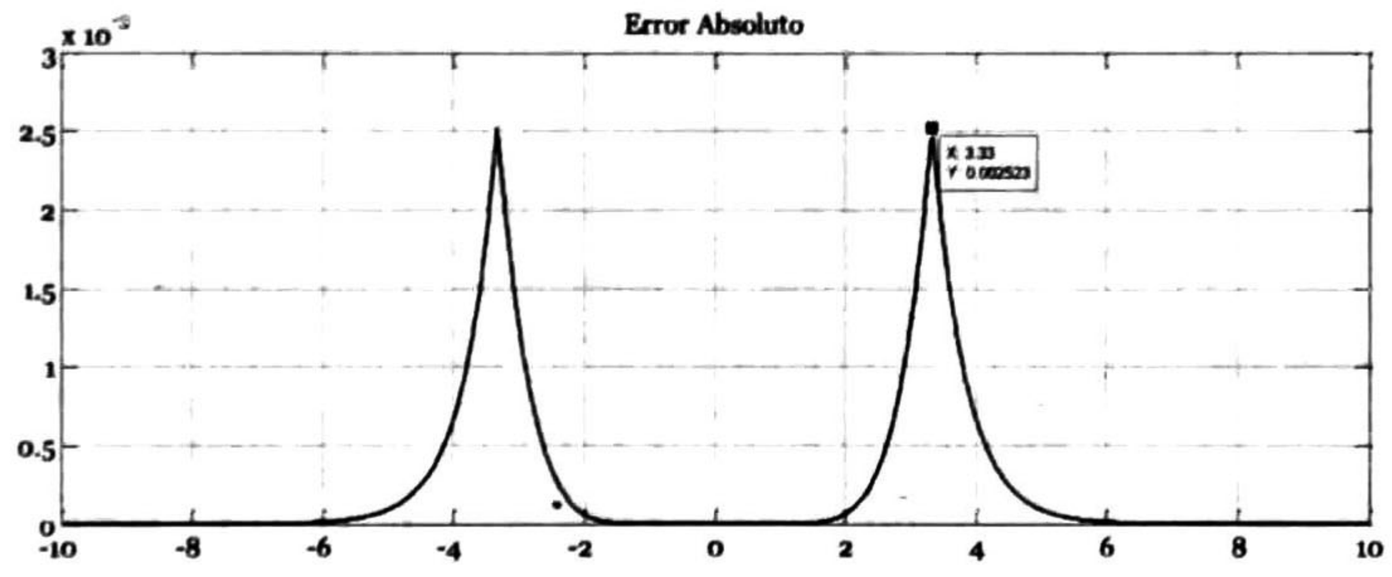


Fig. 4.24 Error absoluto obtenido de las tangentes hiperbólicas.

Como se había mencionado con anterioridad el error máximo del algoritmo obtenido, se genera cuando se encuentra en los límites de los segmentos, el error máximo es de 0.002523 y se puede considerar insignificante.

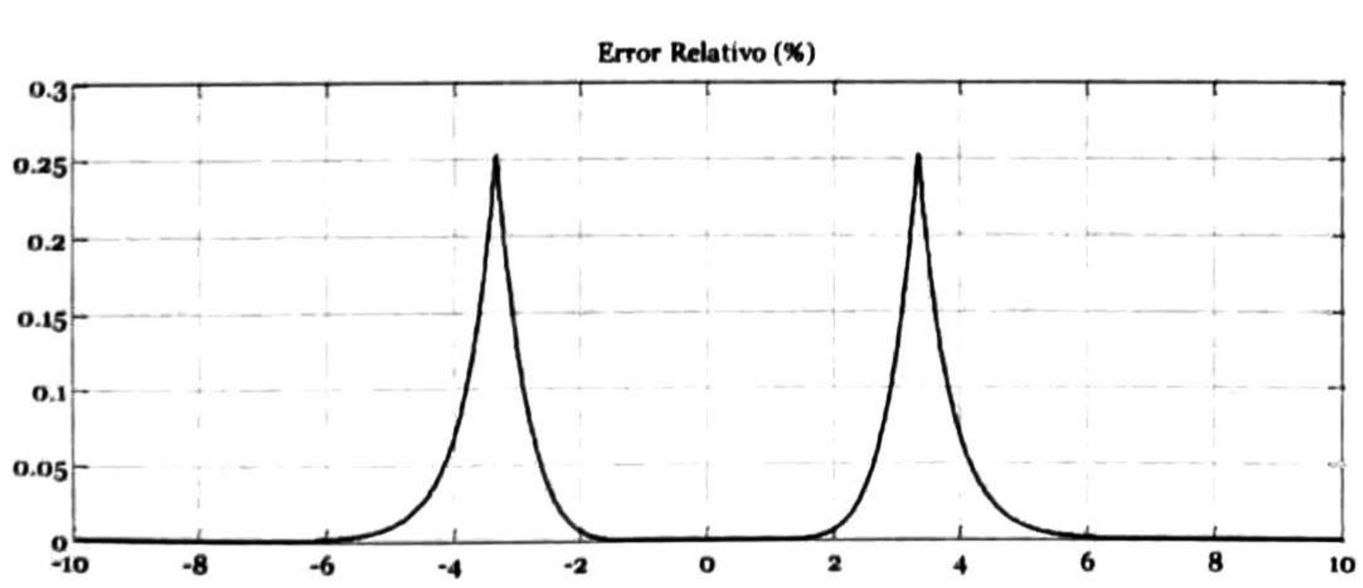


Fig. 4.25 Error relativo porcentual menor al 1%.

4.7 Red neuronal NARX

Una vez que se ha descrito el procedimiento realizado para la implementación de una neurona, es posible desarrollar la arquitectura de la red completa. Entre los parámetros necesarios para el desarrollo se encuentra el número de retardos que poseerá cada una de las entradas, número de neuronas en la capa oculta y capa de salida y los tipos de función de activación de cada una de ellas.

Como se ha mencionado anteriormente la arquitectura NARX requiere de dos tipos de función de activación, que son tangente hiperbólica para la capa oculta y la función lineal para la capa de salida.

La red neuronal NARX ha presentado gran correspondencia con los resultados obtenidos en la siguiente configuración: 4 retardos por entrada, 10 neuronas en la capa oculta y 2 neuronas en la capa de salida.

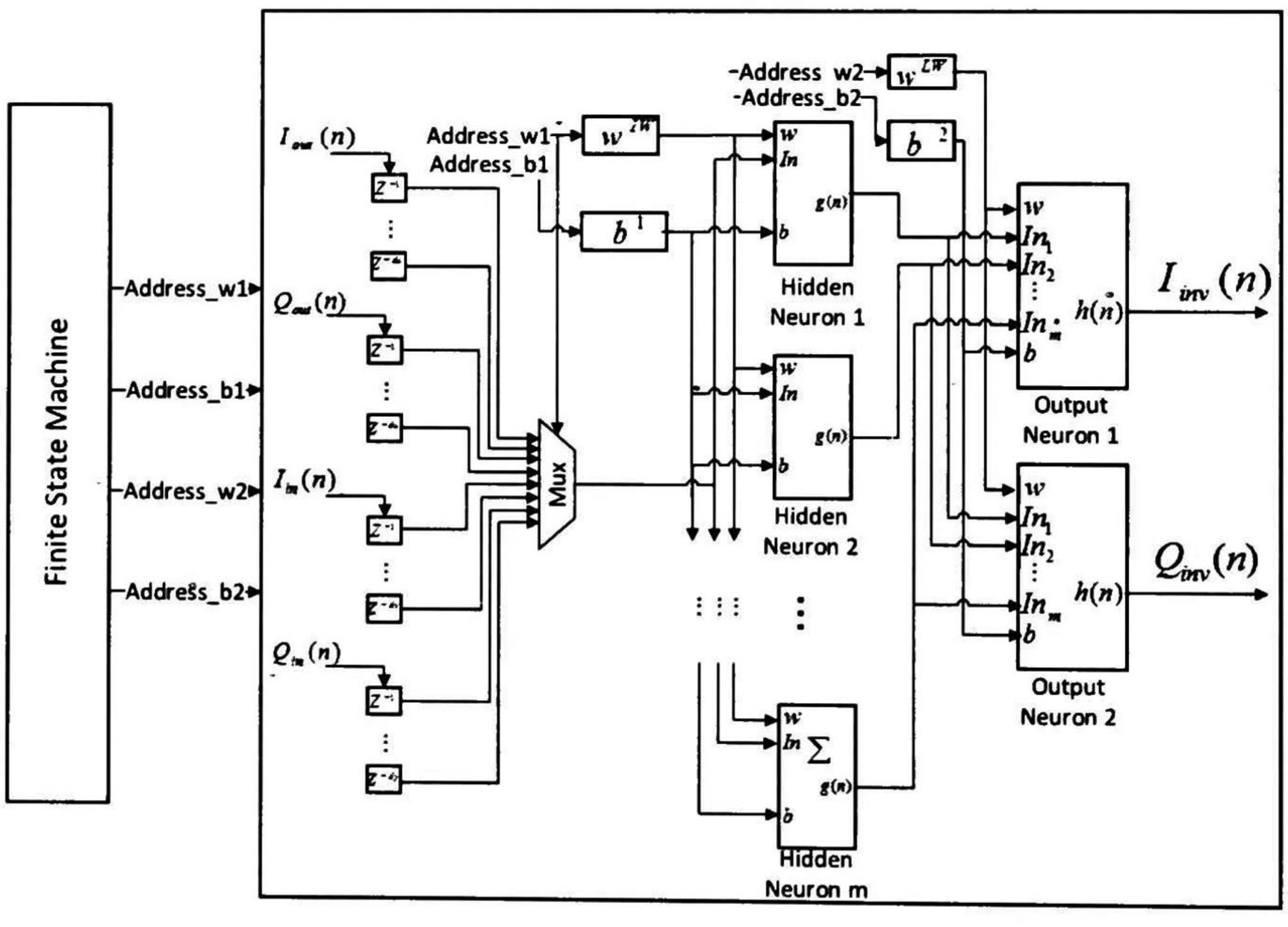


Fig. 4.26 Arquitectura de Red neuronal NARX.

En esta arquitectura se muestra la configuración de la red neuronal que se ha implementado en el dispositivo reconfigurable, el funcionamiento de la arquitectura se realiza de forma secuencial; como ya se mencionó con anterioridad en el diseño de la neurona solo se puede procesar un dato de entrada a la vez, por lo que al colocar un conjunto de 10 neurona en la capa oculta la arquitectura solo puede controlar las 10 primeras multiplicaciones y para ello se ha colocado un multiplexor que selecciona

el dato de entrada a procesar. Las neuronas realizan consecutivamente las multiplicaciones de cada una de las entradas hasta finalizar con ellas, posteriormente se introducen los valores de polarizaciones a cada una de las neuronas, una vez finalizado cada neurona procesa la tangente hiperbólica del valor almacenado en su unidad de procesamiento. Un proceso similar realizan las neuronas de la capa de salida solo que debido a que su función de activación es lineal la sumatoria directamente se muestra en la salida.

Cuando el proceso finaliza, se realiza el ajuste de las entradas moviendo los datos contenidos en los retardos e introduciendo una medición nueva para su posterior procesamiento.

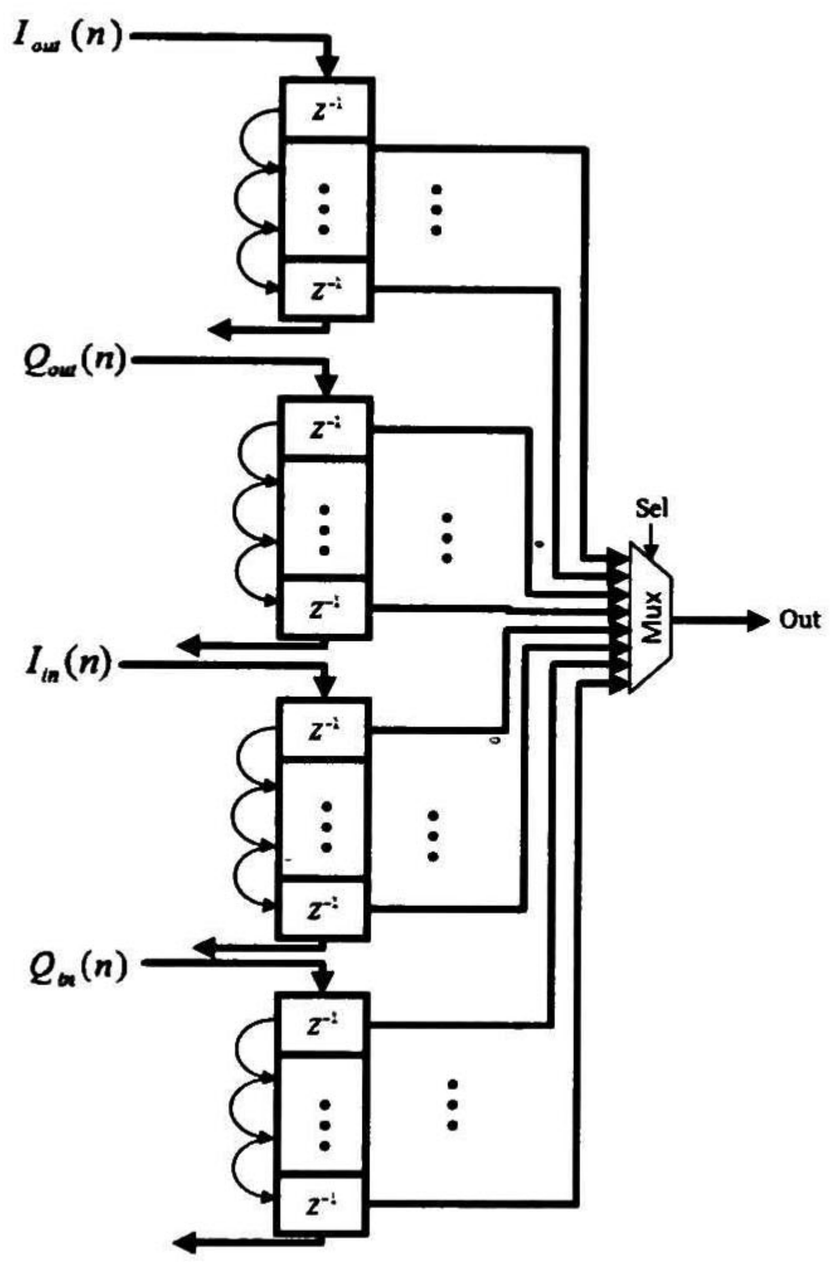


Fig. 4.27 El diagrama muestra la propagación de las entradas y los retardos.

4.7.1 FSM para control de red Neuronal NARX

Una red neuronal debe de estar preparada para poder manejar grandes bloques de información, por lo que el desarrollo de una forma de control es de vital de importación para un óptimo desempeño, en dispositivos reconfigurables es común emplear máquinas de estados finitos para monitorear y activar las señales en los tiempos adecuados. Las máquinas de estados más comunes: tipo Moore donde la salida solo depende del estado actual y es independiente de la entrada. La del tipo Mealy depende del estado, entradas y salidas presentes.

Se ha desarrollado una máquina de estados tipo Mealy que controla los datos de entrada obtenidos de las unidades de almacenamiento, se monitorea las señales de procesos finalizados y el flujo de datos en las neuronas. El diagrama de estados se muestra en Fig. 4.28.

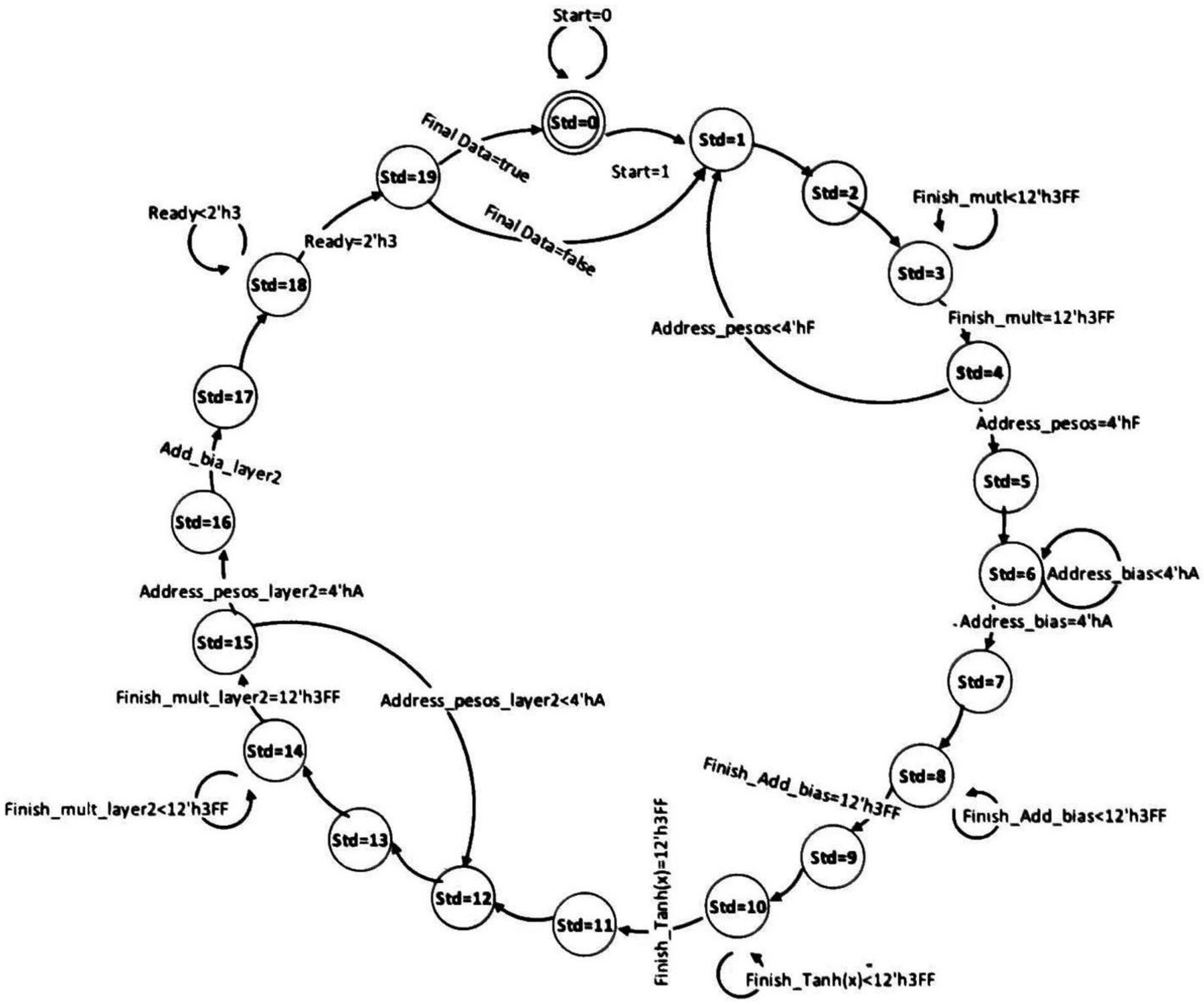


Fig. 4.28 FSM de control para la red neuronal NARX.

La FSM implementada está compuesta por 20 estados, que en conjunto son capaces de controlar la red neuronal que se muestra en Fig. 4.26. La descripción de los estados se describe a continuación:

Stdo. Estado de reposo y de RESET de las variables de la arquitectura, se requiere de un flanco de inicialización para procesar los datos.

Std1. Activa las señales de que inicializan los procesos de las neuronales de la capa oculta.

Std2. Desactiva señales que inicializan las neuronas.

Std3. Realiza la verificación de las señales "Ready" de todas las neuronas de la capa oculta. La máquina no cambia de estado a menos que todas las neuronas hayan terminado de procesar los datos.

Std4. Se aplica un RESET de las señales "Ready", posteriormente verifica que se hayan realizado las multiplicaciones de todas las entradas y retardos. Si no se han finalizado las multiplicaciones la FSM retorna a "Std1", en caso contrario el estado siguiente es "Std5". Se ajustan las posiciones de memoria donde están almacenados los valores de las polarizaciones.

Std5. Se inicia la suma de las polarizaciones.

Std6. Verifica si se han introducido todas las polarizaciones de las neuronas, en caso negativo se espera en este estado hasta finalizar, a continuación se brinca al siguiente estado.

Std7. Restablece las señales empleadas para introducir los valores de las polarizaciones.

Std8. Se verifican que las señales de "Ready" para la suma de las polarizaciones hayan finalizado, en caso negativo se espera hasta finalizar.

Std9. Restablece señales de "Ready".

Std10. Inicia el cálculo de la tangente hiperbólica de cada una de las neuronas de la capa oculta y espera que se haya finalizado para hacer el cambio de estado.

Std11. Restablece señales de fin del cálculo.

Std12. Inicializa el cálculo de la capa de salida.

Std13. Restablece señales.

Std14. Verifica que las señales de "Ready" de las multiplicaciones de la capa de salida haya finalizado.

Std15. Se verifica que todas entradas de la capa de salida se hayan procesado, en caso negativo retorna a "Std12" en caso contrario cambia a "Std16".

Std16. Restablece señales e inicia la suma de la polarización de la neurona 1 de la capa de salida.

Std17. Inicia con la suma de la polarización de la neurona 2.

Std18. Espera a que la señal de "Ready" indique que haya finalizado la suma de las polarizaciones.

Std19. Envía los datos procesados a las salida, y verifica que todos los datos del vector de entrada se hayan procesado, si se han finalizado los datos retorna a "Std0", en caso negativo el proceso se inicia de nuevo en "Std1" con los siguientes datos de entrada.

Capítulo 5

Arquitectura del DPD

En este capítulo se presenta la arquitectura del predistorsionador digital que se ha implementado en FPGA de Xilinx, se hacen uso de los algoritmos diseñados en el Capítulo 4 y se anexan bloques adicionales para su correcto funcionamiento. La arquitectura final está comprendida por el diseño en el FPGA y un bloque adicional que comprende la simulación y validación de los datos obtenidos.

5.1 Introducción

La descripción completa de la arquitectura de un predistorsionador no solo comprende el diseño de una red neuronal aunque es un bloque de suma importancia en el diseño, cabe destacar que se debe de complementar con bloques adicionales que permitan al diseño funcionar de la forma correcta. La arquitectura de un predistorsionador digital implementado en FPGA, debe de contar con algún medio de comunicación con otros dispositivos o un equipo de cómputo, por lo que el desarrollo de un protocolo de comunicación es necesario para el transporte de datos.

Tener un medio de validación y un punto de comparación de los datos obtenidos en FPGA, es lo que permite medir la correspondencia que presentan los resultados y el uso de la herramienta Matlab para el procesamiento matemático, permite implementar una plataforma de comparación de los resultados. El software permite implementar redes neuronales de diversas arquitecturas o en su defecto diseñado a medida, por lo que el uso del *Toolbox* de redes neuronales de Matlab juega un papel importante en la validación de los resultados.

5.2 Uso del Toolbox de redes neuronales de Matlab

Esta herramienta proporcionada por Matlab, permite el diseño de redes neuronales con arquitecturas definidas o a medida, el ajuste de los parámetros de una red neuronal y simulación de datos.

Esta herramienta ha sido empleada para obtener los valores de los pesos y polarización empleados en el FPGA, por lo que el modo de operación del dispositivo reconfigurable es *offline*. El entrenamiento de la red neuronal ha sido a través de Matlab empleando una arquitectura Serie-paralelo, ya que este modo de entrenamiento hace uso de las entradas y salidas para su funcionamiento y muestra gran correspondencia en aplicaciones de sistemas no lineales.

El *Toolbox* de Matlab, permite implementar de forma directa la arquitectura de la red NARX y solo es necesario declarar las características de ella.

Los vectores de entradas deben de estar estructurados en dos filas de idéntico tamaño, donde la columna debe de contener a la pareja de datos reales e imaginarios. Estos vectores no se encuentran en el formato requerido para ser procesados por la red neuronal debido a que estos vectores son concurrentes y se requiere que los datos sean secuenciales para realizar la conversión de formatos se emplea la instrucción "con2seq(x)". Por sintaxis, se definirá a los datos de entrada como vector "u" y los datos de las salidas como vector "y".

```
si_ent(1,:) = PA_salida_i';
si_ent(2,:) = PA_salida_r';
u = con2seq(si_ent);
si_sal(1,:) = PA_entrada_i';
si_sal(2,:) = PA_entrada_r';
y = con2seq(si_sal);
```

Las dos primeras líneas del código anterior ajustan las entradas en dos filas con datos real e imaginario, y la tercera línea realiza la conversión a vectores secuenciales. Posteriormente la definición de los retardos es un factor que le proporciona la característica de poder modelar los efectos de memoria del sistema, para ello se definen dos variables que pueden ser ajustadas con un número de retardos diferentes según el criterio del usuario.

```
du = [1:4];
dy = [1:4];
```

El vector "du" representa los retardos tomados para los datos de entrada, mientras que "dy" los retardos para el vector de salida. Este valor define la cantidad de entradas de inicialización que son introducidas a la red antes de obtener la primera salida consistente. Por lo que el número total de resultados obtenidos será disminuido en la cantidad de retardos que se le hayan asignado a las entradas.

La herramienta de Matlab permite la implementación de la Red NARX con una sola línea de código, ya que es una arquitectura predeterminada y no es necesario definirla por el usuario, para su correcta configuración se requiere definirle los retardos de las entradas y la cantidad de neuronas definidas en su capa oculta. Por inferencia la red neuronal asigna la cantidad de salidas que van en función de la cantidad de filas contenidas en el vector "y"

narx_pd = narxnet(du,dy,12);

La red neuronal puede ser visualizada con la instrucción siguiente y la ventana de la red declarada se muestra en Fig. 5.1.

"view(narx_pd)"

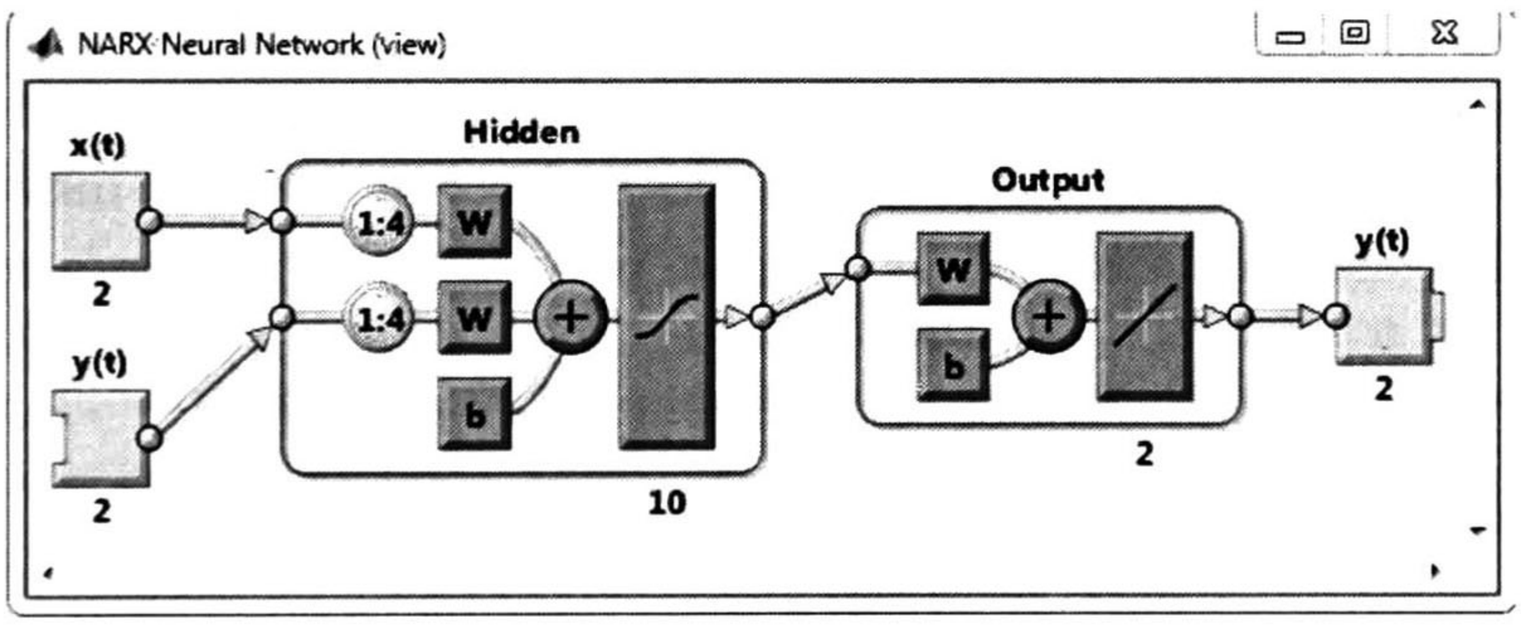


Fig. 5.1 Arquitectura NARX declarada en Matlab.

5.2.1 Entrenamiento de red NARX

La red neuronal NARX ha sido entrenada con algoritmo de Levenberg-Marquardt que ofrece una mayor velocidad de entrenamiento para redes de tamaño moderado. La función tiene una característica de reducción de memoria cuando el conjunto de datos para el entrenamiento es demasiado grande.

Levenberg-Marquardt fue diseñado para aproximar la velocidad de entrenamiento de segundo orden sin tener que calcular una matriz Hessiana. Cuando la función de rendimiento tiene la forma de sumas de cuadrados, entonces la matriz Hessiana puede ser aproximada como:

$$\mathbf{H} = \mathbf{J}^T \mathbf{J} \tag{5.1}$$

y el gradiente puede ser calculado:

$$\mathbf{g} = \mathbf{J}^T \mathbf{e} \tag{5.2}$$

donde **J** es la matriz Jacobiana, el cual contiene las primeras derivadas de los errores de la red respecto a los pesos y las polarizaciones, y **e** es un vector de errores de la red. La matriz Jacobiana puede ser calculada a través de la técnica estándar de *backpropagation*[41].

El algoritmo Levenberg-Marquart emplea la aproximación de la matriz Hessiana y método de *Newton*:

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \left[\mathbf{J}^T \mathbf{J} + \mu \mathbf{I}\right]^{-1} \mathbf{J}^T \mathbf{e} \tag{5.3}$$

Cuando el factor μ es cero, el algoritmo trabaja como el método de *Newton*, empleando la aproximación de la matriz Hessiana. Cuando μ es muy grande, el cálculo del gradiente descendiente se relaliza con pasos pequeños. El método de Newton es más rápido y preciso cerca de un mínimo de error, por lo que el objetivo es desplazarse hacia el newton de lo más rápidamente posible. Así que μ disminuye después de cada paso exitoso y se incrementa sólo cuando se podría aumentar la función de rendimiento.

```
narx_pd.trainParam.mu = 1;
narx_pd.trainParam.mu_dec = 0.8;
narx_pd.trainParam.mu_inc = 1.5;
```

Las líneas anteriores definen los incrementos y decrementos del factor μ y su valor inicial.

Los parámetros importantes para el entrenamiento son el número de épocas, forma de inicialización de parámetros de la red, y la división de los datos para entrenamiento, validación y pruebas.

El número de épocas es un parámetro que detiene el entrenamiento. Una época básicamente es la introducción de los datos empleados para el entrenamiento sin modificar los pesos y las polarizaciones, si algún

---

[41] Hagan, Martin T., and Mohammad B. Menhaj. "Training feedforward networks with the Marquardt algorithm." *Neural Networks, IEEE Transactions on* 5, no. 6 (1994): 989-993.

parámetro como el error mínimo, gradiente mínimo, o error de validación se encuentra dentro de un rango definido por el usuario, el entrenamiento se detiene proporcionando los parámetros de la red que mejor se ajustaron.

```
narx_pd.trainParam.epochs = 500;
```

La función "initnw" es normalmente usada para las redes feedforward que emplean una función tipo sigmoidea. Esta técnica de inicialización de los datos está basada en Nguyen and Widrow[42] que genera pesos y polarización iniciales.

```
narx_pd.layers{1}.initFcn = 'initnw';
narx_pd.layers{2}.initFcn = 'initnw';
```

Las siguientes instrucciones definen la forma en que se dividen los datos introducidos en la red neuronal, ya que pueden ser empleados para entrenamiento, validación y prueba de la red.

```
narx_pd.divideParam.trainRatio = 40/100;
narx_pd.divideParam.valRatio = 40/100;
narx_pd.divideParam.testRatio = 20/100;
```

La función "Preparets" ajustan las parejas de los vectores consecutivos empleados para el entrenamiento, así como también definen un nuevo vector que contiene los elementos de inicialización de los retardos de la red.

```
[p, Pi, Ai, t] = preparets(narx_pd,u,{},y);
```

Posteriormente la red puede ser entrenada con la siguiente instrucción.

```
narx_pd = train(narx_pd,p,t,Pi,Ai);
```

La Fig. 5.2 muestra la ventana de entrenamiento, donde se muestra la arquitectura de la red implementada y la evolución de los para parámetros de entrenamiento.

---

[42] Nguyen, Derrick, and Bernard Widrow. "Improving the learning speed of 2-layer neural networks by choosing initial values of the adaptive weights." In Neural Networks, 1990., 1990 IJCNN International Joint Conference on, pp. 21-26. IEEE, 1990.

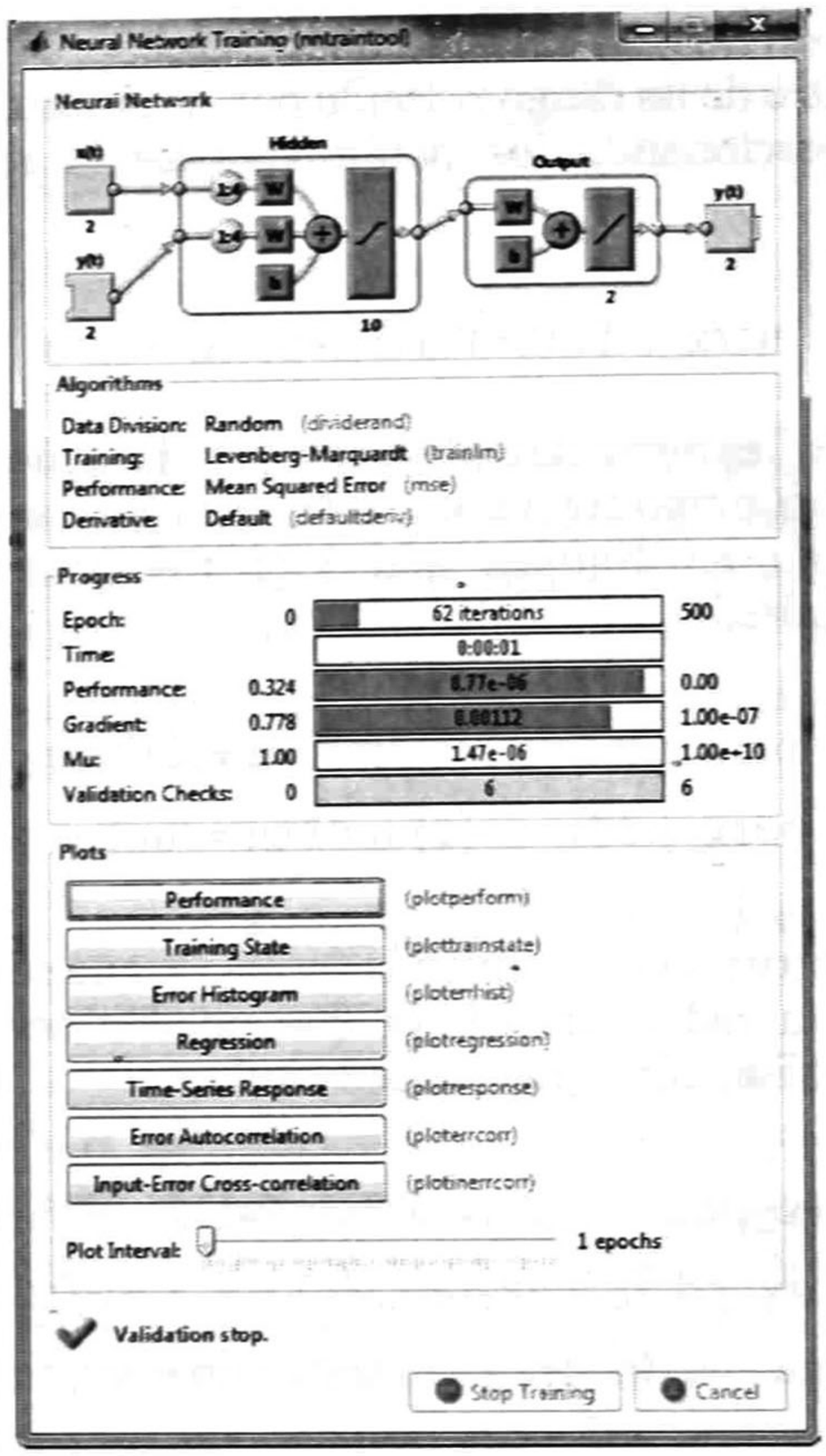


Fig. 5.2 Ventana de entrenamiento de redes neuronales en Matlab.

Una vez finalizado la red puede simular los datos con los valores de pesos obtenidos en el entrenamiento, los pesos de ambas capas y las polarizaciones pueden ser matrices definidas por el usuario.

```
y_narx_pd = narx_pd(p,Pi,Ai);
pesosi = narx_pd.IW;
pesosl = narx_pd.LW;
bias = narx_pd.b;
```

5.3 Preprocesamiento y postprocesameinto

El entrenamiento de una red neuronal puede ser más eficiente si se realizan ciertos pasos de pre-procesamiento de las entradas y los objetivos. Por ejemplo en una red multicapa con una función sigmoidea que generalmente es usada en la capa oculta. La función llega a saturarse cuando la entrada es mayor a 3. Si esto sucede cuando el proceso de

entrenamiento está iniciando, los gradientes pueden ser muy pequeños y el entrenamiento de la red puede ser muy lento. En la primera capa de la red, la entrada es un producto de las entradas de tiempo por los pesos y la adición de la polarización, si la entrada es muy grande, entonces los pesos deben ser muy pequeños con la finalidad de evitar que la función se sature, por lo que es una práctica estándar normalizar los datos entes de aplicarse a la red[43].

Generalmente, el paso de normalización es aplicado tanto a los vectores de entrada y los objetivos, de esta forma la salida de la red, siempre cae dentro de un rango normalizado. Cuando la red es puesta en campo, es necesario un proceso transformación inversa para retornar a las unidades originales.

De forma práctica se puede decir que la red presenta un bloque de pre-procesamiento y post-procesamiento de datos.

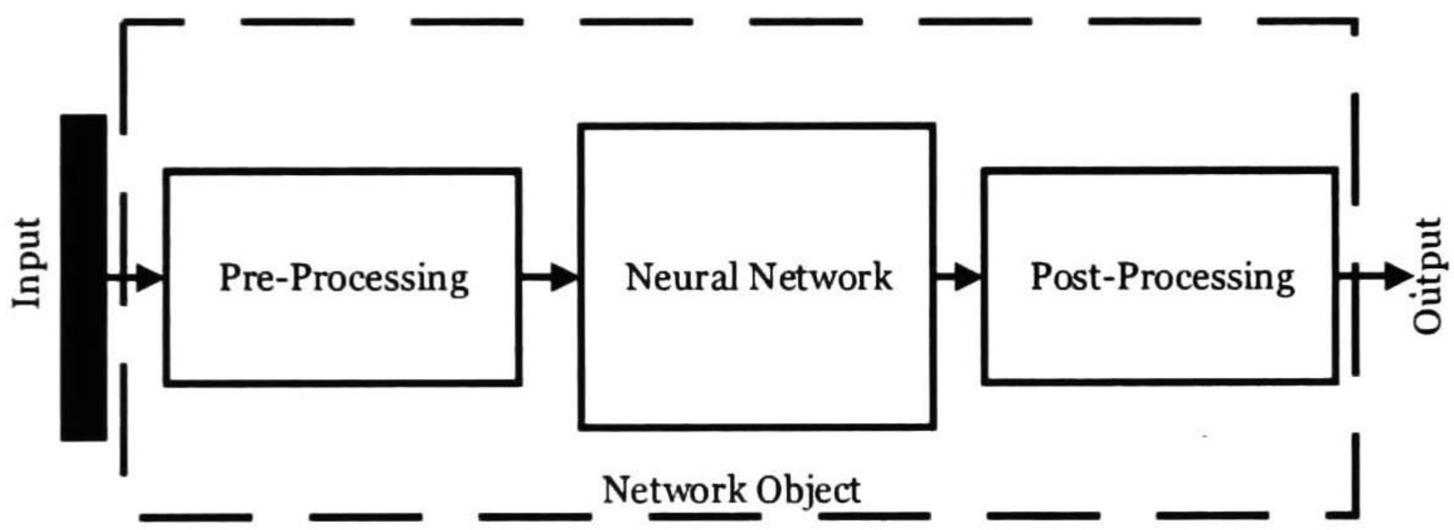


Fig. 5.3 Procesamiento de datos.

La función de normalización está dada por:

$$y_{norm} = \frac{(y_{\max} - y_{\min}) * (x_{in} - x_{\min})}{(x_{\max} - x_{\min})} + y_{\min} \tag{5.4}$$

donde y_{min} y y_{max} corresponden al rango de normalización ±1, x_{min} y x_{max} son los valores máximos y mínimos del vector a normalizar y x_{in} es el dato a procesar.

Para la desnormalización de los datos, se hace un procedimiento algebraico de la función mostrada en (5.4), son el valor a calcular es x_{in}.

---

[43] Neural Network Toolbox, User's Guide R2014a", www.mathworks.com/help/pdf_doc/nnet/nnet_ug.pdf, The MathWorks.Inc, Pg.2-9, 2-10, 2-11.

5.4 Comunicación RS-232

Para realizar comunicación serial entre el FPGA y el equipo de cómputo, es necesario una interfaz que permita empatar los niveles lógicos, la *Virtex-6 FPGA ML605 Evaluation Kit* incorpora el chip CP2103[44] que permite realizar un puente de USB-UART virtual con una amplia variedad de tasas de transferencia. El FPGA tiene pines dedicados para la comunicación y recepción de la comunicación serial, por lo que solo es necesario mandar la cadena de datos que se desea transmitir.

La implementación de transmisión de comunicación serial asíncrona fue desarrollada por medio de una máquina de estados, que transmite bit a bit la cadena de datos obtenida como resultados de la red neuronal.

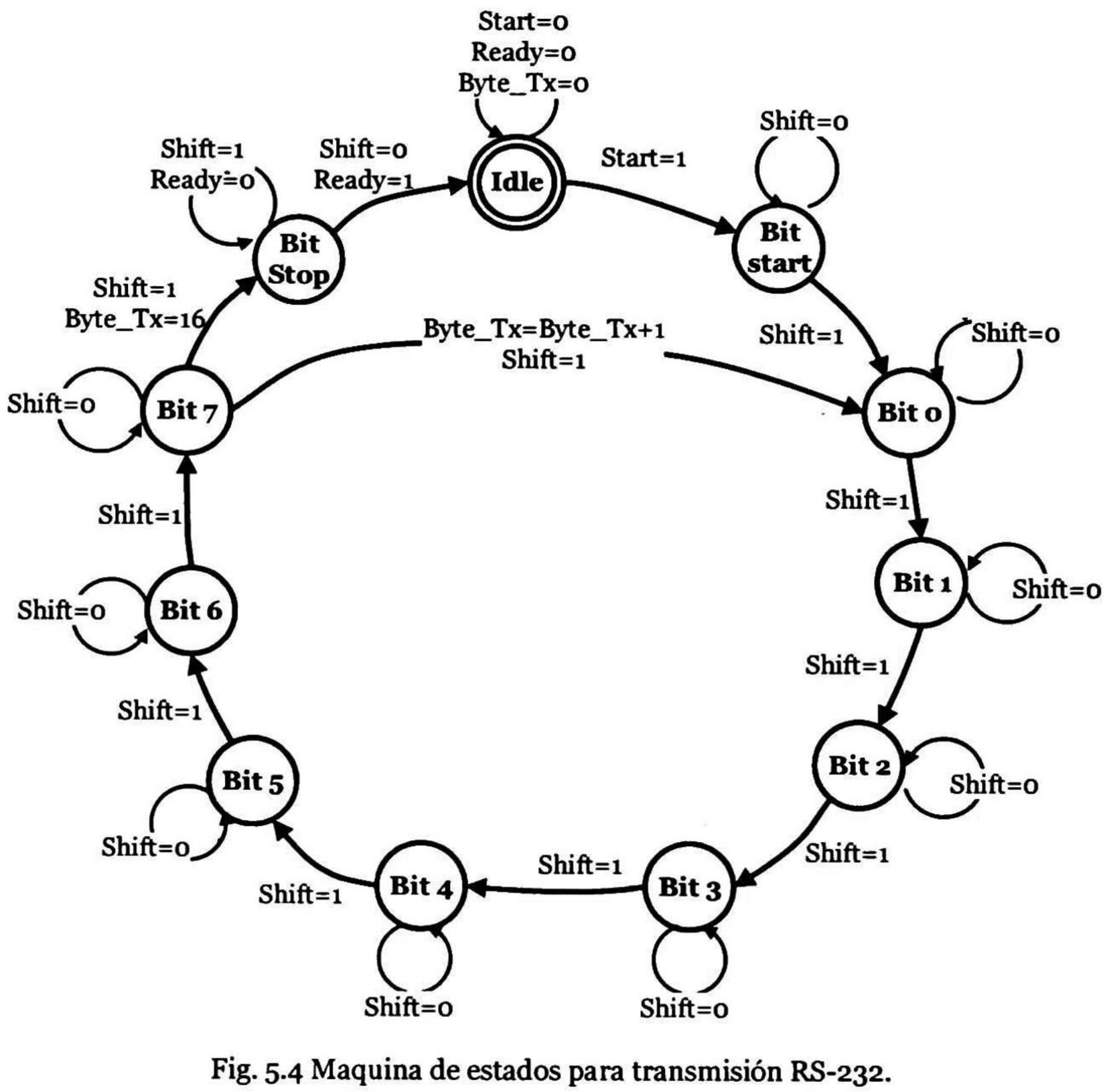


Fig. 5.4 Maquina de estados para transmisión RS-232.

El proceso de transmisión tiene la finalidad de empatar los formatos que permite la herramienta de Matlab, después se realiza una conversión de precisión simple a doble, y posteriormente cada nibble contenido en dato de precisión doble es convertido a ASCII. Por lo que se transmiten 16 bytes de información por dato procesado.

---

[44] http://www.silabs.com/products/interface/usbtouart/Pages/usb-to-uart-bridge.aspx

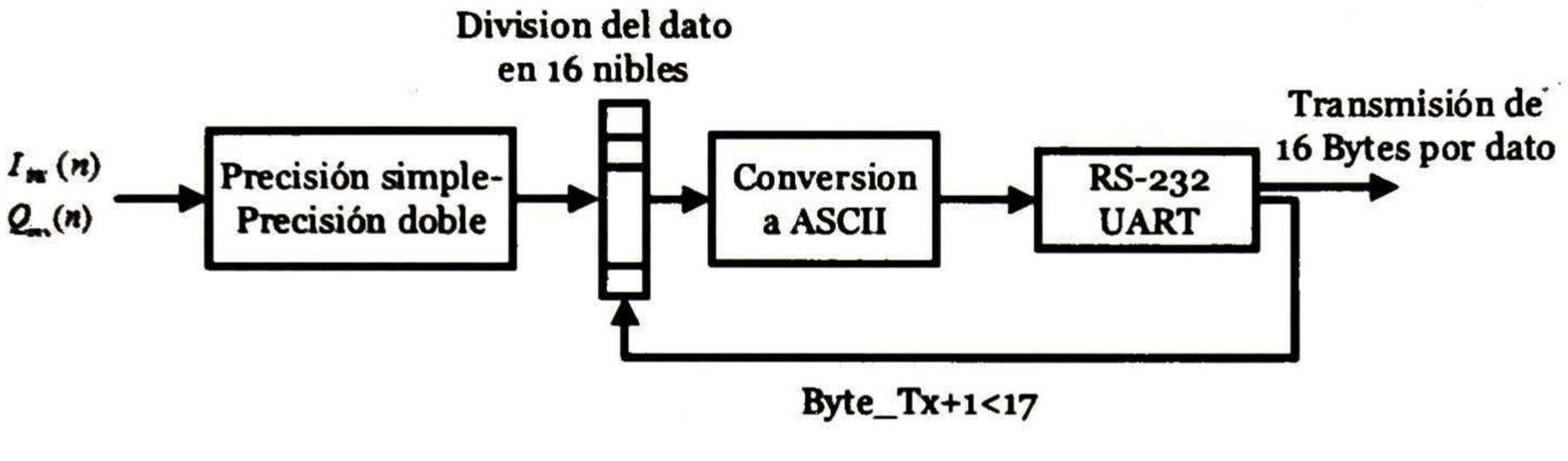


Fig. 5.5 Diagrama a bloques de transmisión de datos.

5.5 Arquitectura del predistorsionador digital

La arquitectura final comprende etapas de normalización y desnormalización, procesamiento, comunicación y validación.

El proceso de normalización es una técnica que permite escalar un conjunto de datos a otro conjunto de datos, comúnmente empleado cuando el rango de valores es muy grande y se pretende trasladar a un conjunto reducido. El proceso de desnormalización es la operación inversa a la normalización y esta se encarga de trasladar un vector normalizado a sus unidades originales.

El bloque de procesamiento es el cálculo matemático realizado por la red neuronal, éste tiene como función obtener la predicción de salidas con un conjunto de entradas presentes, la comunicación es la interfaz que permite transmitir los resultados del FPGA con el equipo de cómputo para su posterior validación.

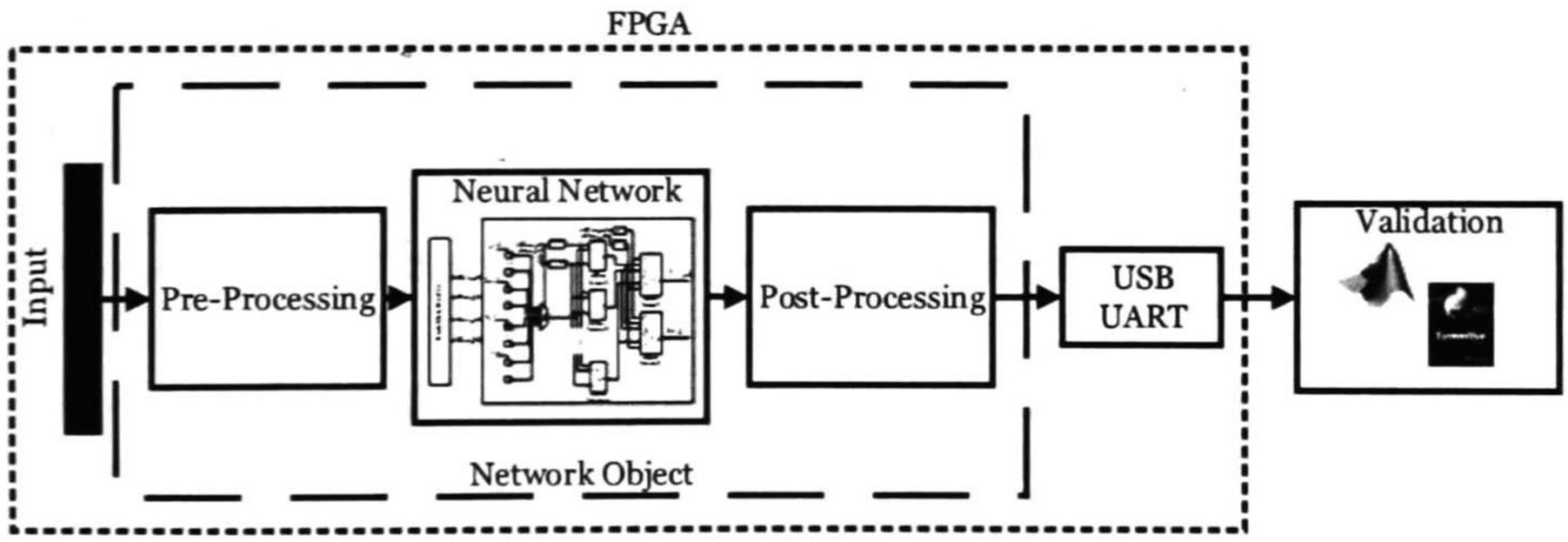


Fig. 5.6 Predistorsionador digital implementado.

La arquitectura es totalmente secuencial, por lo que es necesario que cada proceso haya finalizado para la inicialización del siguiente proceso, el flujo del proceso se realiza de izquierda a derecha como se describe en el diagrama a bloques de Fig. 5.6.

En la **Tabla 5.1** se muestra los recursos utilizados por el FPGA al implementar la arquitectura del predistorsionador digital así como la frecuencia máxima de operación.

Tabla 5.1 Recursos utilizados por el predistorsionador digital.

Resource utilization	Hidden Neuron	Output Neuron	Multiplier	Hyperbolic Function	Adder	Total Used	Available
Slice Registers	3699	672	97	3027	578	38572	301440
Slice LUTs	**2697**	**573**	**96**	**2131**	**446**	**29057**	**150720**
LUT FF	2116	461	79	1645	354	21491	89013
Bonded IOBs	**137**	**135**	**100**	**68**	**68**	**3**	**600**
Block RAM						225	416
DSP	**6**	**2**	**2**	**4**	**2**	**64**	**768**
Maximum Operation Frequency	95.511 MHz						

Capítulo 6

Simulación y resultados

En este capítulo se realizan las simulaciones de los bloques empleados para la implementación del predistorsionador digital, y se hacen pruebas al sistema implementado en el FPGA, los resultados obtenidos son comparados con Matlab y simulados en la herramienta SystemVue que es una herramienta empleada para el diseño de RF. Ésta permite visualizar las características de predistorsión.

6.1 Introducción

La simulación de los bloques que se implementan en un dispositivo reconfigurable, es un paso muy importante en el flujo de diseño ya que nos muestra de manera amplia el comportamiento que tendrá el sistema al introducirle un conjunto de entradas conocidas, de esta manera se puede comprobar si el bloque funcionará de manera correcta o en su defecto si requiere modificaciones.

Las simulaciones que se pueden emplear en la herramienta de Xilinx, pueden ser de manera "*Behavioral*" que simula de manera ideal el funcionamiento de los bloques, la simulación "*Post-Translate*" que traslada el lenguaje de descripción de hardware en funciones para el FPGA, el "Pos-Map" simula la función una vez que ha sido mapeada de acuerdo a las características del FPGA, por último y más importante la simulación "Post-Place&Route" que es la simulación más cercana al comportamiento real del FPGA que implementa las funciones mapeadas incluyendo los retardos de las conexiones, hace uso de parámetros que pueden influir en un comportamiento en campo del sistema.

6.2 Simulaciones

La implementación de una multiplicación por medio de una máquina de estados, influye directamente en la cantidad de ciclos de reloj a emplear para obtener una salida consistente, debido a que este tipo de algoritmo requiere de un conjunto de operaciones básicas que son ejecutadas en un mínimo de 5 ciclos de reloj.

Si multiplicamos dos datos cualesquiera y sus equivalentes en hexadecimal, podemos multiplicarlos y comparar los resultados.

Tabla 6.1 Datos de prueba para el algoritmo de multiplicación.

Datos	Decimal	Hexadecimal
Dato A	5.245	40a7d70a
Dato B	**0.1254**	**3e009d49**
Multiplicación	0.658772	3f28a548

En la **Tabla 6.1** se presentan dos datos cualesquiera y sus equivalentes en hexadecimal empleados para la simulación.

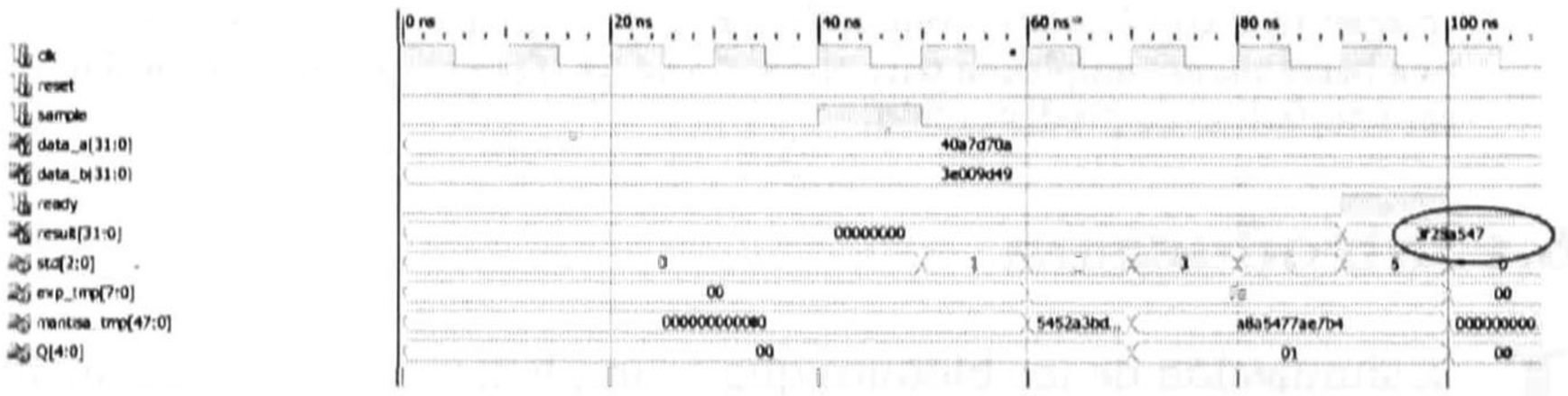


Fig. 6.1 Simulación del algoritmo de multiplicación.

Como se muestra en la simulación, una vez que se encuentran presentes lo datos hexadecimales en las entradas del bloque, es necesario un flanco para inicializar el bloque.

La sumatoria es la combinación de un *CORE Generator* de Xilinx que permite realizar sumas y restas en coma flotante (*Floating Point*) y un bloque acumulador programado en Verilog. Las sumas son realzadas cada vez que se presenta la entrada del dato y activa el flanco de inicialización.

Tabla 6.2 Datos empleados en la sumatoria.

Data A	Suma 1 (hex)	Suma 2 (hex)	Suma 3 (hex)	Suma 4 (hex)	Suma 5 (hex)
0.25d/3e800000h	3e800000	3f000000	3f400000	3f800000	3fa00000

La simulación realiza 5 sumas sucesivas de una entrada con valor de 0.25.

Fig. 6.2 Simulación de la sumatoria.

La tangente hiperbólica empleada en la capa oculta es una función de activación no lineal, y el algoritmo implementado que se ha mostrado en Fig. 4.22 aplica un método hibrido por segmentos y aplicación de un polinomio. Este algoritmo consume una mayor cantidad de ciclos de reloj en comparación de métodos alternativos, pero esto es compensado con el aumento de precisión y la capacidad de procesar cualquier dato en la entrada.

Tabla 6.3 Tabla de simulación de tangente hiperbólica.

Fase de entrada	Tanh (Maltab)		Tanh (FPGA)	
	(Hex)	(decimal)	(Hex)	(decimal)
-4.5 (c0900000)	bf7fefd4	-0.9997532	bf800000	-1
0.75 (3f400000)	3f22991f	0.6351489	3f229920	0.6351489
4.5 (40900000)	3f7fefd4	0.9997532	3f800000	1

En la **Tabla 6.3** se evalúan los tres segmentos del método hibrido implementado para obtener la tangente hiperbólica.

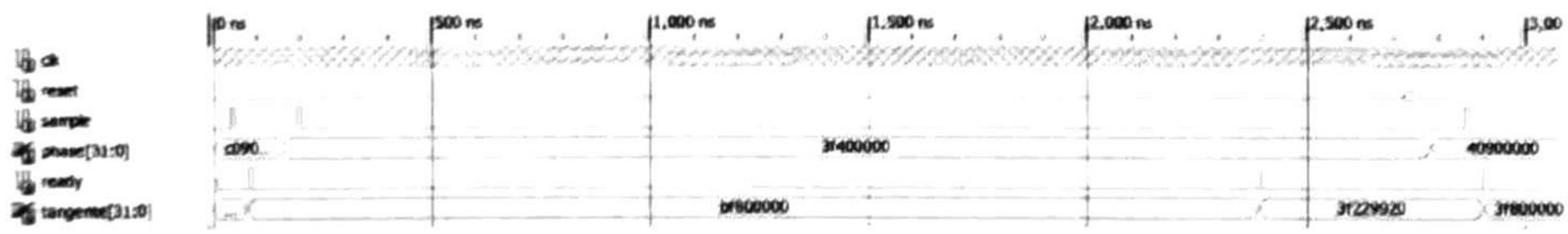


Fig. 6.3 Simulación de tangente hiperbólica.

La neurona artificial es la unidad básica empleada en la redes neuronales, y esta combina sumatorias, multiplicaciones y funciones no lineales para obtener una salida. La cantidad de ciclos requeridos para obtener una salida, depende directamente de la cantidad de entradas y retardos contengan la neurona, ya que esto incrementa el tamaño de la matriz de pesos y la cantidad de sumas empleadas. La función de la neurona es introducir cada una de las entradas y multiplicarlas por su peso correspondiente, los resultados obtenidos son almacenados en la sumatoria para posteriormente calcular la función de activación. La

simulación de la red NARX realiza cuenta con 180 multiplicaciones, 192 sumas y 10 tangentes hiperbólicas. Que es la cantidad de operaciones necesarias procesar 2 entradas real-imaginaria retardada 4 posiciones cada una de ellas en un conjunto de 10 neuronas en la capa oculta y 2 en la capa de salida.

Fig. 6.4 Simulación de la Red NARX.

El protocolo RS-232, como se ha mencionado con anterioridad permite realizar comunicación entre el FPGA y el equipo computo, para realizar la transmisión de un dato obtenido de la red neuronal, se requieres del envión de un conjunto de 16 bytes, que es el equivalente de un dato en coma flotante de doble precisión en código ASCII.

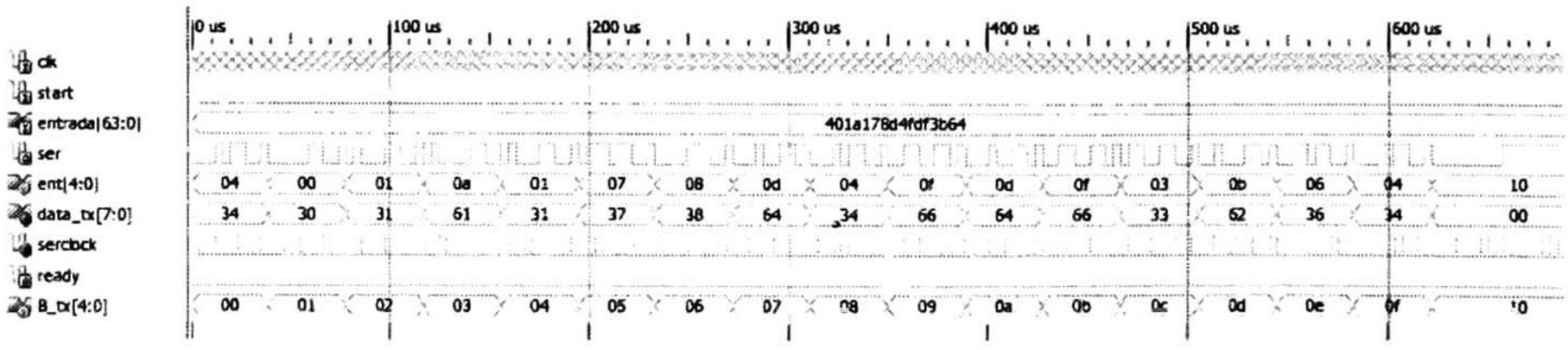


Fig. 6.5 Simulación de la comunicación Rs-232.

6.3 Resultados

El desarrollo experimental con un amplificador de potencia fue usado para validar la implementación de la red NARX en FPGA. Una señal LTE a 2 GHz es generada en un software de RF llamado SystemVue y descargada a un generador de señales vectoriales. El amplificador de potencia es conectado a un atenuador de 30 dB. Las entradas y salidas son medidas con un analizador de espectros empleando la herramienta 89600VSA.

Los datos medidos por el analizador de espectros son procesados por la red NARX y son enviados por medio de comunicación asíncrona (UART) al equipo de cómputo para ser almacenados para su posterior comparación y validación. Para realizar la recepción de los datos en el equipo de cómputo, es necesario realizar un "script" que permite la configuraron del puerto, como son velocidad de transferencia, número máximo de bytes que puede recibir en una sola lectura y el comando de fin de transmisión.

De igual forma se implementa la red NARX con el Toolbox de redes neuronales de Matlab y se realiza la simulación de los mismos datos de entrada empleados en el FPGA.

Tabla 6.4 Datos obtenidos por la Red NARX en el FPGA y Matlab.

Results	Real(FPGA)	Image(FPGA)	Real(Matlab)	Image(Matlab)
1	-0.34510845317838	0.07818963943093	-0.34510342853533	0.07818700472594
2	-0.35257565491600	-0.02967927526307	-0.35257891077726	-0.02967838693830
3	-0.24657515530991	-0.07807854950230	-0.24664802091774	-0.07809724509940
4	-0.06395308502237	-0.03039482902367	-0.06402700042977	-0.03042185011335
5	0.12606102178778	0.07410325482868	0.12605209565504	0.07410446812629
6	0.25096877400903	0.17675751911770	0.25096190357323	0.17675773608755
7	0.28196600764153	0.20763142923692	0.28196250978712	0.20762731818684
8	0.21321615559612	0.15994817431051	0.21321632703128	0.15994433789154
9	0.12060241480182	0.06695307107320	0.12060398021669	0.06695367249790
10	0.03242863820595	0.00092798982272	0.03242695709790	0.00092780150537

Los resultados obtenidos por ambos métodos de procesamiento son comparados y las diferencias más significantes se encuentran en los decimales menos significativos, y el principal factor en la variación de estos resultados, se debe a que la arquitectura implementada en el FPGA trabaja totalmente en formato de precisión simple y solo realiza un conversión de formato para la transmisión de datos, mientras que Matlab trabaja totalmente en precisión doble lo que afecta directamente en los bits menos significativos del dato. Calculado el error obtenido en los resultados es menor a 1%, por lo que la arquitectura presenta un alto grado de confiabilidad en sus resultados.

Al realizar la simulación de los resultados en SystemVue, se ha comprobado que la red NARX es adecuada para el modelado no lineal con excelentes resultados en el modelado de las características inversas (AM/AM, y AM/PM) de los amplificadores de potencia.

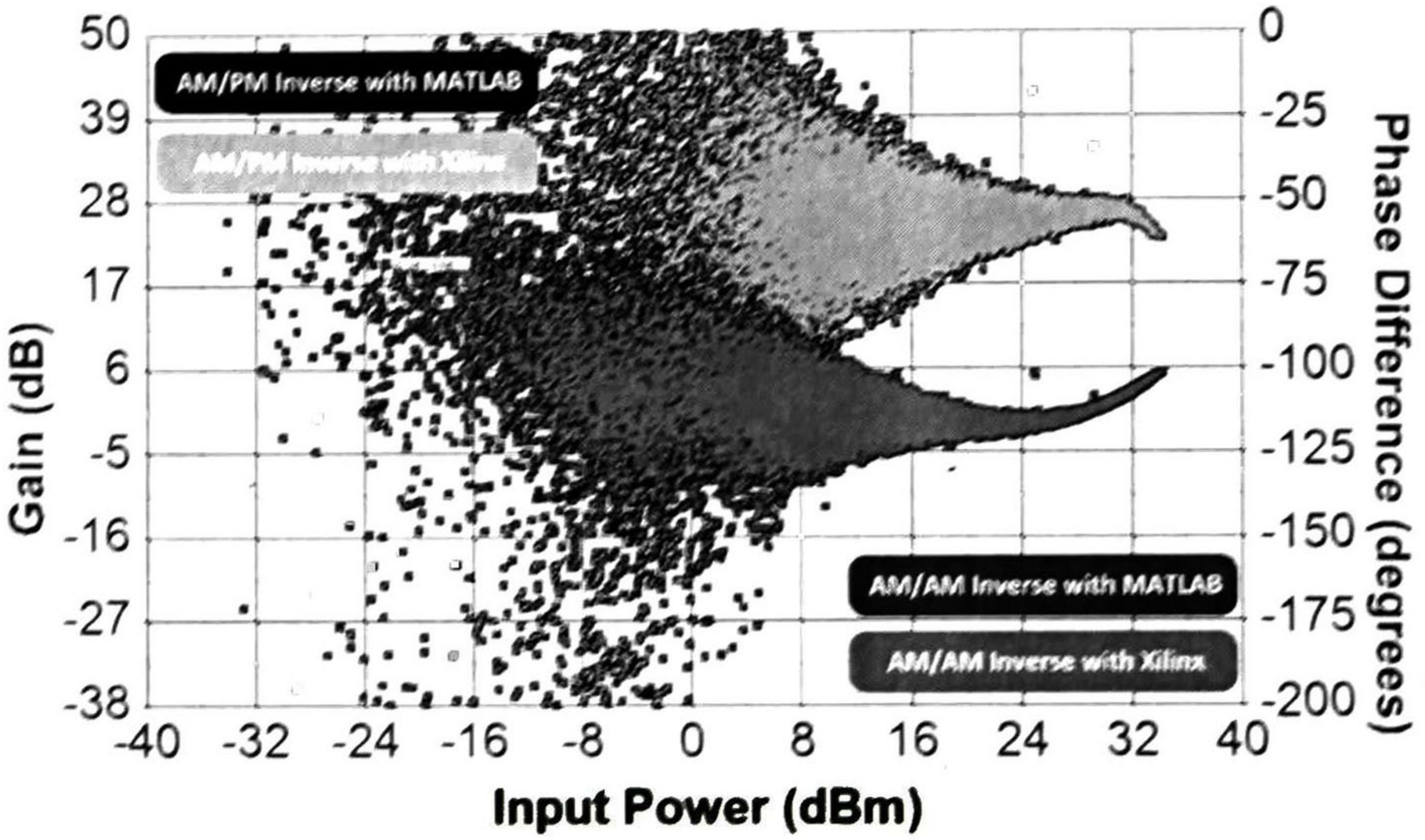


Fig. 6.6 Simulación de las características inversas del amplificador de potencia.

Capítulo 7

Conclusiones y trabajos futuros

En este capítulo se presentan las conclusiones del trabajo de tesis, así como los trabajos futuros que se pueden desarrollar a partir de él. También se incluyen las publicaciones que resultaron del desarrollo de este documento.

7.1 Conclusiones

En este trabajo de tesis se desarrolló la implementación de una red neurona NARX utilizando un dispositivo reconfigurable FPGA de Xilinx, se desarrolló la misma arquitectura empleando el Toolbox de redes neuronales de Matlab y así tener un punto de comparación de los resultados en ambos métodos de procesamiento. Se ha comprobado que la aplicación de redes neuronales para el modelado de sistemas no lineales presenta gran ventaja en comparación de métodos tradicionales, ya que éste al ser un sistema adaptivo, las variaciones generadas por factores externos en la señal de entrada, no muestra problema debido a la característica de generalización que presentan las redes neuronales.

El nuevo método de implementación de la tangente hiperbólica, sacrifica ciclos de reloj por el aumento de precisión y rango de entrada de datos que puede procesar, quedando limitado a la precisión permitida por el formato coma flotante de simple precisión.

En este trabajo se realiza el procesamiento de los datos antes de ser introducidos a la red neuronal, lo que permite que el entrenamiento se realice con mayor rapidez y evitando caer en un problema de no convergencia del entrenamiento.

La red neuronal se ha desarrollado para trabajar totalmente secuencial, lo que presenta una desventaja ante arquitecturas pipeline o segmentadas ya que carece de algún tipo de método de aceleración.

En la **Tabla 6.4** ha muestra la gran correspondencia que presenta el sistema con respecto a Matlab, y los resultados obtenidos solo varían en los decimales menos significativos ganando el sistema un alto grado de confiabilidad de sus resultados.

Finalmente se puede concluir que la aplicación de la red neuronal NARX para el modelado de sistemas no lineales con efectos de memoria, ha mostrado ser un método exitoso debido a la gran correspondencia de sus resultados

7.2 Trabajo Futuro

El presente proyecto de tesis puede ser mejorado y ampliado mediante alguna de las siguientes actividades.

- Implementar algún tipo de segmentación, que permita acelerar el flujo de datos entre la entrada y la salida.

- Realizar una arquitectura con código genérico, que permita declarar la red neuronal solo empleando parámetros de configuración.

- Realizar un algoritmo de entrenamiento en FPGA, y así eliminar el uso del equipo de cómputo para el procesamiento de los datos de la red. El algoritmo propuesto para su validación es un filtro de Kalman ya que en desarrollos propuestos por investigadores ha presentado buena correspondencia de resultados y su implementación emplea operaciones matemáticas de complejidad moderada.

- Implementación del sistema en modo *online*.

7.3 Publicaciones realizadas

7.3.1 Publicación

J.A. Renteria-Cedano, L.M. Aguilar-Lobo, J.R. Loo-Yau, S. Ortega-Cisneros, "Implementation of a NARX Neural Network in a FPGA for Modeling the Inverse Characteristics of Power Amplifiers", *IEEE* 57th International Midwest Symposium on Circuits And Systems, August 2014.

7.3.2 Publicación enviada y en revisión.

J.A. Renteria-Cedano, L.M. Aguilar-Lobo, J.R. Loo-Yau, S. Ortega-Cisneros, J.J. Raygoza Panduro "FPGA Implementation of a NARX Network for Modeling Nonlinear Systems", Countdown for the 19th Iberoamerican Congress on Pattern Recognition , November 2014.

Implementation of a NARX Neural Network in a FPGA for Modeling the Inverse Characterisctics of Power Amplifiers

J. A. Renteria-Cedano, L. M. Aguilar-Lobo, J. R. Loo-Yau, S. Ortega-Cisneros
[1]Departamento de Ing. Eléctrica y Ciencias Computacionales
Centro de Investigación y de Estudios Avanzados del I.P.N.
Unidad Guadalajara, Av. Del Bosque 1145, Colonia El Bajío.
C.P. 45019, Zapopan, Jalisco, México
email: jrenteria@gdl.cinvestav.mx, laguilar@gdl.cinvestav.mx

Abstract — **In this paper the hardware implementation of a NARX neural network algorithm using a Field Programmable Gate Array (FPGA) is presented. A NARX network is a Recurrent Neural Network (RNN) suitable for modeling nonlinear systems with promising results for the modeling of the inverse characteristics (AM/AM and AM/PM) of Power Amplifier (PAs). The implementation is realized in XilinX ISE tool with the Virtex-6 FPGA ML 605 Evaluation Kit using Verilog language. Experimental results have shown a high correlation with the inverse model computed with SystemVue in co-simulation with MATLAB for a GaN class F PA working with a LTE signal center at 2 GHz.**

Keywords *— ANN, Verilog, Xilinx, FPGA, NARX network.*

I. INTRODUCTION

Artificial Neural Networks (ANNs) have been studied extensively over many years. One of the most important conclusions with ANNs is that a feedforward network is sufficient for approximating any measurable function on a compact domain. Recently, Recurrent Neural Networks (RNNs) have been developed as a solution to model dynamic systems. These RNNs have several advantages that have been explained in previous works [1]-[2].

There are several types of RNNs, for example: distributed time delay neural network (TDNN) [3], layer recurrent network [3] and NARX [4]. This last type of RNNs has been demonstrated that converges much faster and generalizes better than other types of RNNs. The NARX network is a powerful class of RNN, which is well suited for modeling nonlinear systems and is computationally equivalent to a Turing Machine [4].

Currently, we are proposing that the NARX can be used for linearizing microwave power amplifiers (PAs) using the Digital Pre-Distortion technique (DPD). This technique is based on the idea of modified the baseband signal according to the inverse function of the PA [5]. This is performed by a predistorter element located before the PA. Therefore, it is necessary to obtain the inverse model of the PA. This means that the AM/AM and AM/PM [5] characteristics of the PA are required. The inverse model is obtained by the training of the NARX network with measurements signals of the PA.

We have successfully implemented a DPD system based on the NARX network, using software environment (SystemVue and MATLAB), obtaining promising results. The next step is the hardware implementation of the NARX network algorithm. To this is selected a Field Programmable Gate Array (FPGA) because is possible realize a custom design, which only is used the required hardware. As future works, we claim implement the NARX neural network in a processor core of XilinX.

To the best of the author's knowledge, there are not publications regarding to the FPGA implementation of a NARX network. Some publications of FPGA implementation are realized to Hopfield neural network [6] and RVTDNN [7].

This paper describes the implementation of the NARX network architecture in a FPGA using Verilog language in Xilinx ISE Tool with the Virtex-6 FPGA ML605 Evaluation Kit [8]. In order to validate the implementation, the inverse model of a PA will be generated in the FPGA and compared to the one obtained in a co-simulation of SystemVue with MATLAB. The experimental results show a high correlation with those computed in MALTAB.

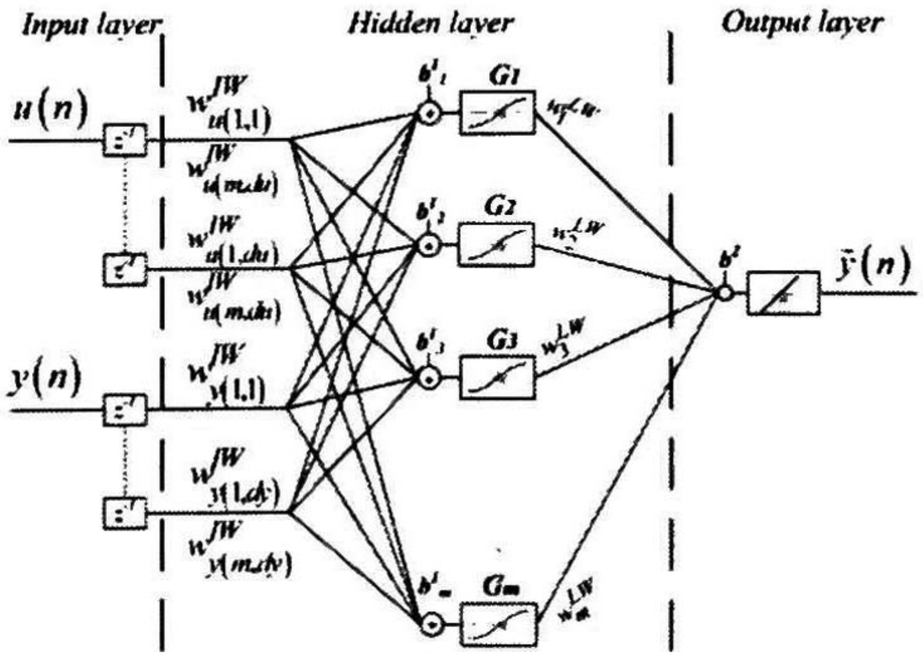


Fig. 1. Neural architecture of the NARX network.

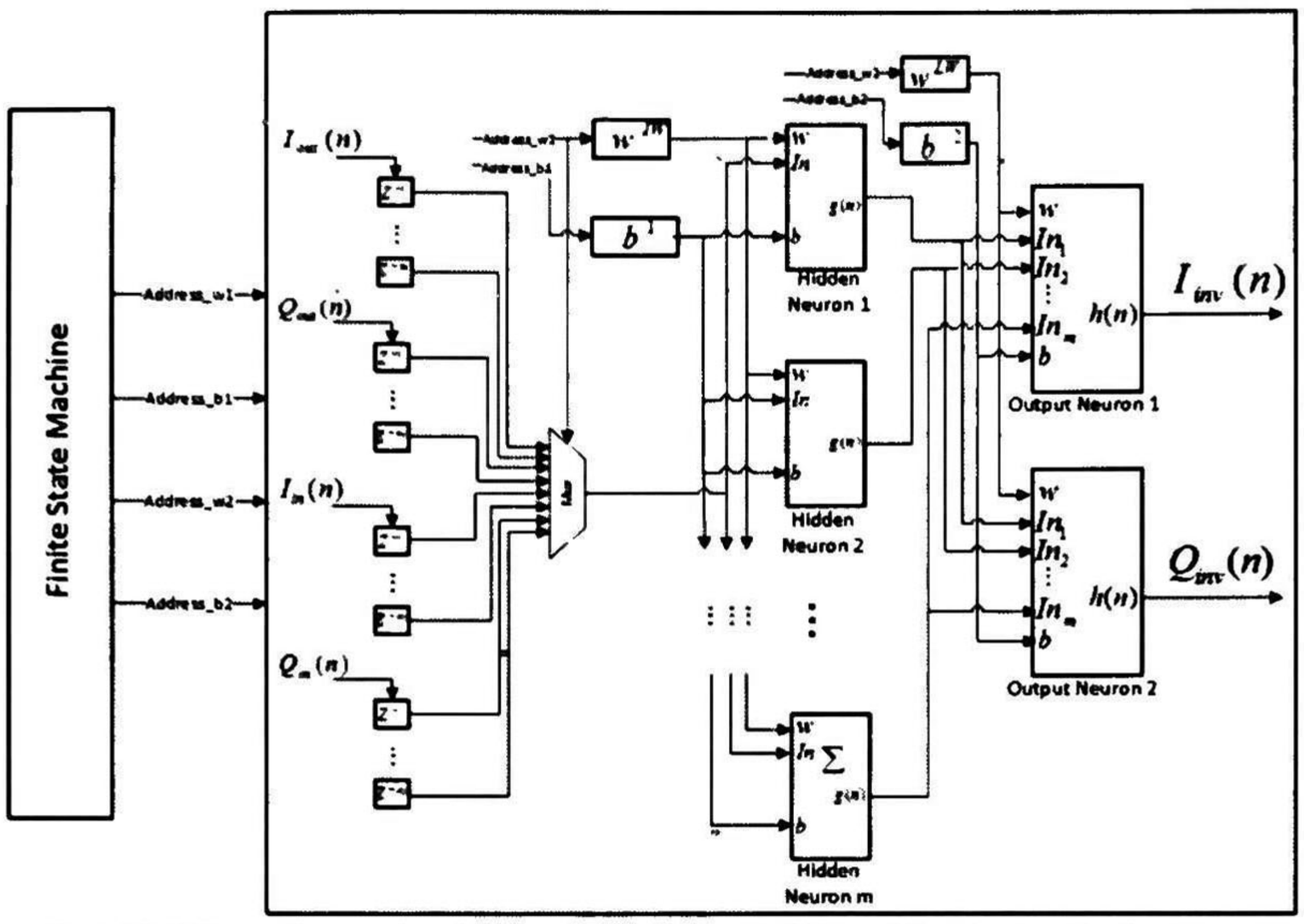


Fig.2 NARX neural network architecture implemented in Xilinx ISE tool using Verilog language.

II. NARX NEURAL NETWORK

The NARX network is a class of RNN which takes as inputs a window of past input and output values to compute the current output [9]. These values are represented by tapped delay lines in the input and output of the system. Mathematically, the output of the NARX network is defined as:

$$\tilde{y} = f\left[u(t-1), u(t-2), \ldots u(t-du), y(t-1), y(t-2), \ldots y(t-dy)\right] \quad (1)$$

The architecture of the NARX is composed of three layers, as is show in Fig. 1. The input layer is formed by the input and output signals of the system to be modeled, *u(t)* and *y(t)* respectively, along with a set of tapped delay lines for each signal. The hidden layer is a set of neurons with a sigmoidal activation function. Each neuron performs the sum of the dot product of the input with the weight matrix and the bias. The result is the argument of the activation function given by the *tansig* function, which can be approximated as a hyperbolic tangent (*tanh*). The output data of each neuron is defined as:

$$G_k(n) = \tanh\left(X_k(n)\right) \quad (2)$$

where

$$X_k(n) = \sum_{j=1}^{du} w_{(k,j)}^{IW} u(n-j) + \sum_{j=1}^{dy} w_{(k,j)}^{IW} y(n-j) + b_k^1 \quad (3)$$

Where $k = 1,2,..m$, and represent the number of neurons in the hidden layer. The output layer is the estimated output; this is obtained by the weighted sum of the output of the hidden layer and the biases with a linear activation function. The outputs data of the NARX network is found:

$$\tilde{y}(n) = \sum_{k=1}^{m} w_k^{LW} G_k(n) + b^2 \quad (4)$$

The embedded memory characteristic of the NARX network represents a significant advantage with respect to other RNNs. These tapped delay lines at the network output help to converge faster and generalize better. In addition, in problems that have long-term dependencies, the performances of the NARX network are much better than other conventional RNNs. The explanation for this behavior is that the output memories of the NARX network can be manifested as jump ahead connections in the time unfolded network [9].

III. FPGA IMPLEMENTATION

The complete neural architecture was implemented in Xilinx ISE Tool. After the training process, the weights and biases matrices obtained were stored in a memory unit.

The implemented structure of the NARX network is shown in Fig. 2. The complete architecture consists of all the operations, such as the weights connections, multipliers, sums and activations functions to the input, hidden, and output layers. The neuron architecture is realized with the scheme presented in [10] and is shown in Fig. 3. This architecture has six main blocks, such as RAM memory to store the weights and biases, a

multiplexer to select the inputs samples, a multiplier, and an adder with an accumulator in floating point and an activation function block.

The basic operation that each neuron performs is initiated by the selection of the input sample and its corresponding 4 bit memory address of the weights matrix. This process is realized for all the connections between the inputs and the neurons in the hidden layer. Then, the multiplication of the connections is performed in parallel form for each neuron; the results are added and accumulated to all the connections. Next, the bias is added to each result of the sum. When the last bias is added, the result in each neuron is passed through the activation function block. In this block the hyperbolic tangent for the neurons in the hidden layer is performed, while neurons of the output layer transfer only the same value to the output.

Hyperbolic tangent function is implemented through a Taylor series, given by the polynomial function

$$\tanh(x) = \frac{362880x + 60480x^3 + 3024x^5 + 72x^7 + x^9}{362880 + 181440x^2 + 15120x^4 + 504x^6 + 9x^8} \quad (5)$$

All these operations are given in floating point with simple precision. The resources of Xilinx used are multiplier blocks to realize the products and CORE for the sums and divisions. The control of the neural architecture such as the address bits, clock, enable signals, are generated by a Finite State Machine (FSM) type Mealy machine [11], shown in Fig. 4. The output values are determined by both its current state and current inputs. The FSM has 20 states that perform 180 products, 192 sums, and 10 hyperbolic tangents, executed in 595 clock cycles.

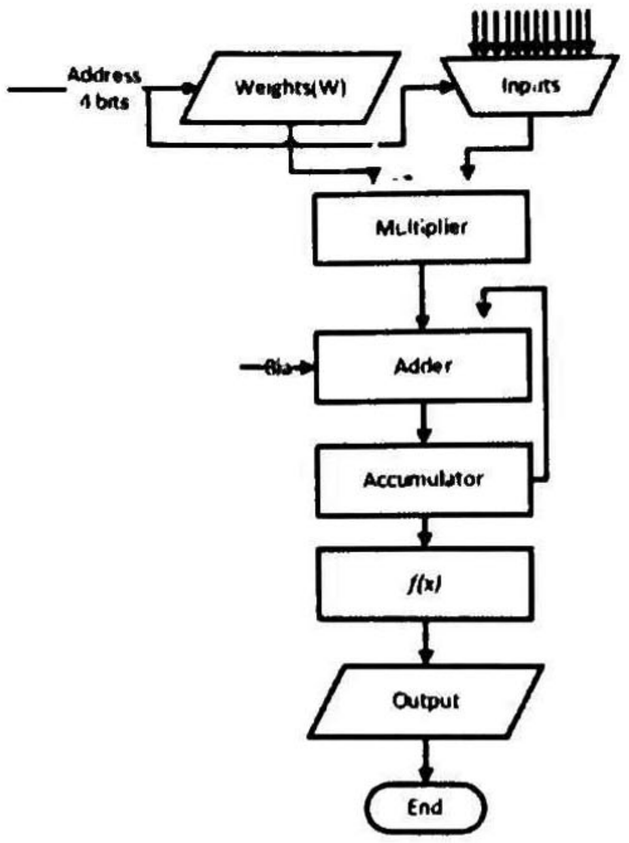


Fig. 3. Neuron architecture implemented in Xilinx ISE tool with Verilog language.

IV. VALIDATION AND RESULTS

An experimental setup with a PA is used to validate the FPGA implementation of the NARX network algorithm. A LTE signal at 2 GHz is generated in SystemVue and downloaded in a Vector Signal Generator to feed the PA. At the PA output a 30 dB attenuator is connected. Input and output signals are measured with a spectrum analyzer using the 89600VSA software tool. The signal processing is performed in the SystemVue and the AM/AM and AM/PM characteristics of the PA are obtained, as shown in Fig. 5.

To this application, the inputs of the NARX network are the in-phase *(I)* and quadrature *(Q)* components of the complex signals at the output and input of the PA, with its tapped delay lines respectively. Therefore, the estimated output of the NARX are the *(I)* and *(Q)* components of the inverse model of the PA.

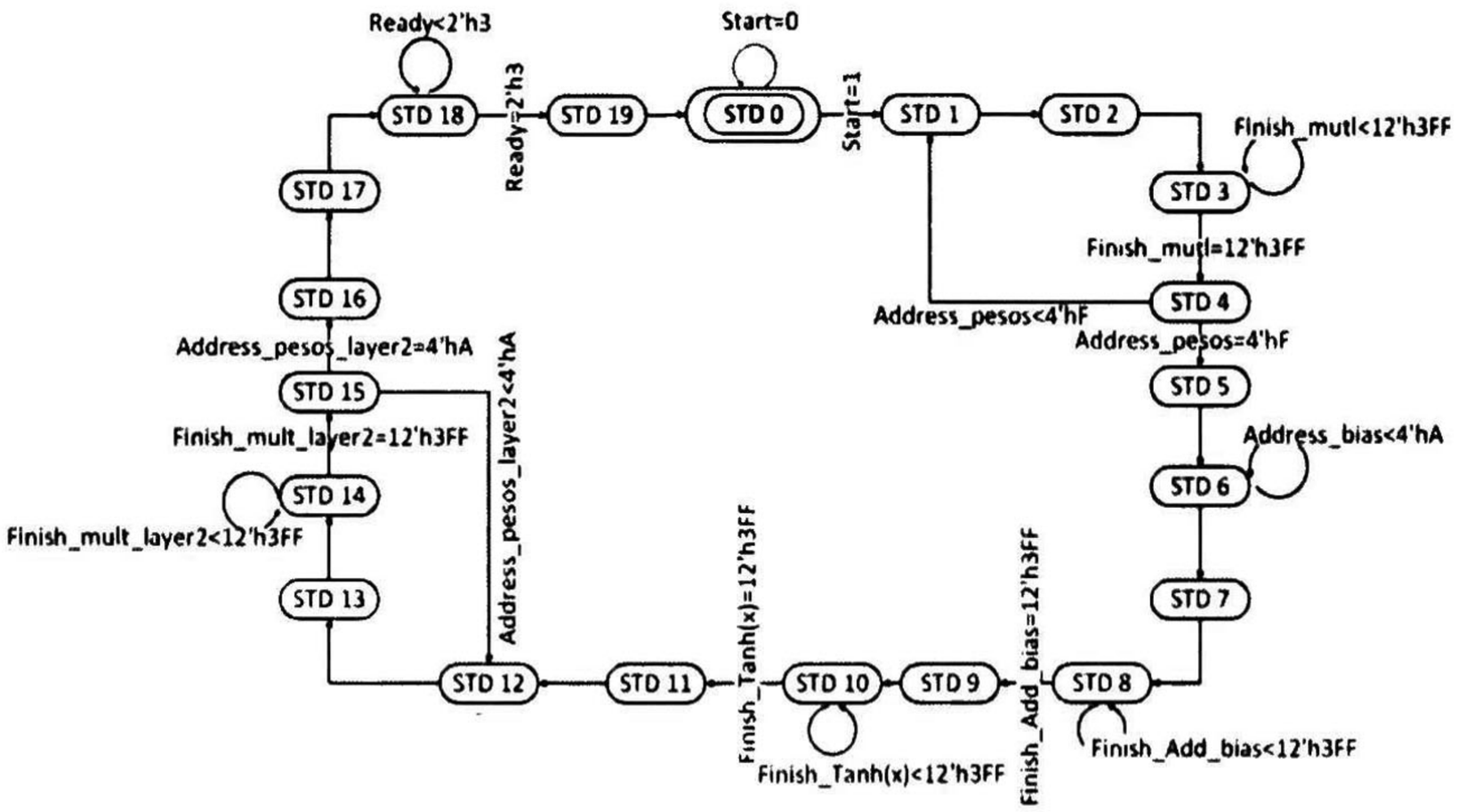


Fig.3. FSM type Mealy implemented in Verilog to control the NARX neural architecture.

With the data sampled collected, a NARX network is trained to obtain the inverse model of the PA. The training is performed with the Neural Network toolbox of MATLAB in a co-simulation environment with SystemVue. In the training process, the best performance of the inverse model is obtained with 4 delays and 10 neurons.

The neural architecture of the trained NARX was implemented in Xilinx ISE Tool (see Fig. 2) with Verilog language. Table I gives the resource requirement reported by Xilinx in details [13]. After simulation the AM/AM and AM/PM inverse characteristics are obtained and compared with the measurements, in order to demonstrate the validity of the implementation. Results are shown in Fig. 6.

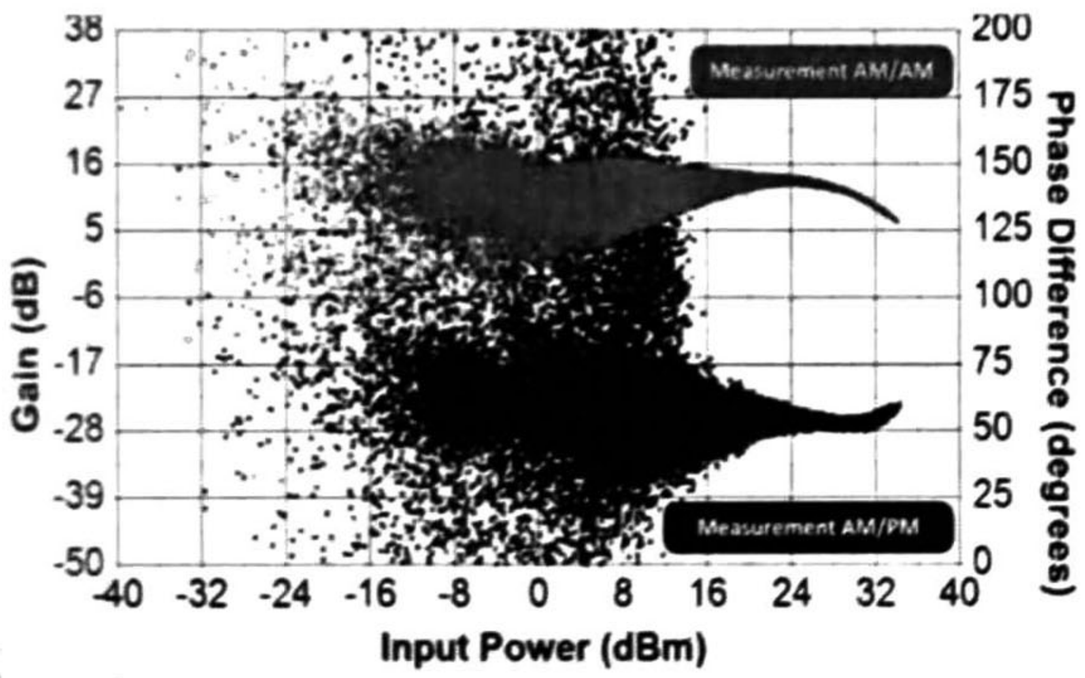


Fig. 5. AM/AM and AM/PM characteristics of PA measurements.

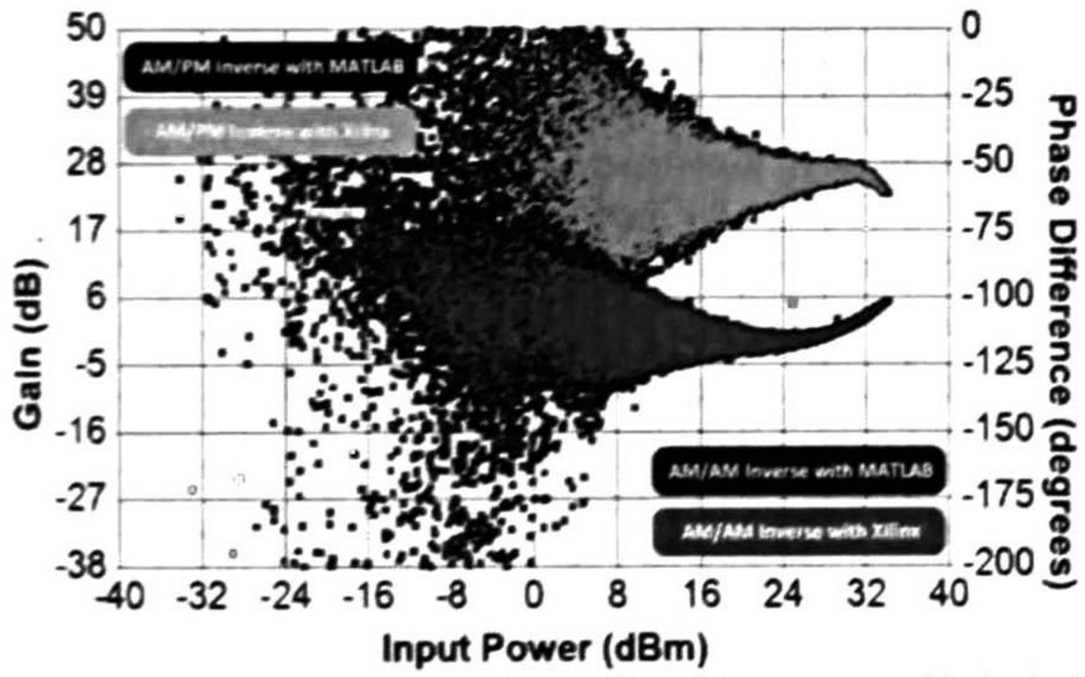


Fig. 6. AM/AM and AM/PM characteristics of PA of the inverse model obtained with the NARX with the MATLAB simulation and with FPGA implementation in Xilinx.

CONCLUSIONS

This paper describes the FPGA implementation of the NARX network in Xilinx ISE with Verilog language. The NARX network has a neural architecture with tapped delay lines at the input and output that help to converge faster and generalize better than others RNNs. The implementation is validated with the modeled of the inverse model of a PA as part of DPD system. The results obtained are compared with the simulation environment and show a high correlation with the FPGA results.

REFERENCES

[1] A. M. Schafer, H. G. Zimmermann, "Recurrent neural networks are universal approximatores", *ICANN 2006*, Part I, *LNCS 4131*, pp. 632640, 2006.

[2] D. Seidl, R. Lorenz, "A structure which a recurrent neural networks can approximate a nonlinear dynamic system", *Proceedings of the International Joint Conference on Neural Network* 1991, Vol. II, pp. 709-714.

[3] S Haykin, "Neural Netowrks: A comprehensive foundation", 2nd Ed. Prentice Hall, 1998.

[4] H. T. Siefelmann, B. G. Horne, C. L. Giles, "Computational capabilities of recurrent NARX neural networks", *IEEE Trans. On Systems, Man and Cybernetics- Part B: Cybernetics*, Vol. 27, No.2, Apr. 1097.

[5] M. Rawat, Karun Rawat and F. M. Ghannouchi,"Adaptative Digital Predistortion of Wireless Power Amplifiers/Transmitters using Dynamic Real-Valued Focused Time-Delay Line Neural Network", *IEEE Trans. Microw. Theory Tech.*, Vol. 58, No. 1, pp. 95-104, Jan. 2010.

[6] M. Atencia , H. Boumeridja, G. Joya, F. García-Lagos, and F. Sandoval, "FPGA implementation of a systems identification module based upon hopfield networks". *Neurocumputing*, 70(2007) 2828-2635, 2007.

[7] M. Bahoura and C.-W. Park, "FPGA-Implementation of an Adaptative Neural Network for RF Power Amplifier Modeling", in *New Circuits and Systems Conference (NEWCAS). 2011 IEEE 9th International, june 2011, pp.29-32.*

[8] "Virtex-6 FPGA ML605 Evaluation Kit", www.xilinx.com/ products/ boards-and-kits/EK-V6-ML605-G htm, Xilinx Inc.

[9] T. Lin, B. G. Horne, P. Tiño anc C. Lee Giles, "Learning long-term dependencies is not difficult with NARX recurrent neural networks", *IEEE Trans. Neural Netowks*, Vol.7 No. 6, 1996.

[10] K. Mohamad, M. F. O. Mahmud, F H Adnan, W. F. H. Abdullah, "Design of single neuron on FPGA", *IEEE Symposium on Humanities, Science and Engineering Research"*, June 2012.

[11] Z.Salcic, A. Smailagic, "Digital system design and prototyping using field programmable logic", Boston: Kluwer Academic Publishers,1997, pp 134-141.

[12] A.L.S. Braga, C.H. Llanos, D. Gohringer, J. Obie, J Becker, M. Hubner, "Performance, Accuracy, Power Consumption and Resource Utilization Analysis for Hardware/Software realized Artificial Neural Networks", *Bio-inspired Computing: Theories and Aplications (BIC-TA), 2010 IEEE Fifth International Conference on*, pp.1629-1636. Sept.2010

[13] *XST User Guide for Virtex-6, Spartan-6, and 7 Series Devices*, 14.1 ed, Xilinx.Inc, April 24 2012, pp 273-285.

TABLE I. RESOURCE UTILIZATION AND MAXIMUM OPERATING FREQUENCY OF THE VIRTEX 6

Resource utilization	Hidden Neuron	Output Neuron	Multiplier	Hyperbolic Function	Adder	Total Used	Available
Slice Registers	3699	672	97	3027	578	38572	301440
Slice LUTs	2697	573	96	2131	446	29057	150720
LUT FF	2116	461	79	1645	354	21491	89013
Bonded IOBs	137	135	100	68	68	3	600
Block RAM						225	416
DSP	6	2	2	4	2	64	768
Maximum Operation Frequency	95.511 MHz						

FPGA Implementation of a NARX Network for Modeling Nonlinear Systems

[1]J.A. Rentería-Cedano, L.M. Aguilar-Lobo, S. Ortega-Cisneros, J.R. Loo-Yau,
[2]J.J. Raygoza Panduro

[1]Departamento de Ing. Eléctrica y Ciencias Computacionales
Centro de Investigación y de Estudios Avanzados del I.P.N.
Unidad Guadalajara, Av. Del Bosque 1145, Colonia El Bajío.
C.P 45019, Zapopan, Jalisco, México.

[2]Departamento de Electrónica
Centro Universitario de Ciencias Exactas e Ingenierías
Universidad de Guadalajara.

Email: jrenteria@gdl.cinvestav.mx,
laguilar@gdl.cinvestav.mx,

Abstract. This paper presents the FPGA implementation of NARX neural network. The complete neural architecture was implemented with Verilog language in Xilinx ISE Tool with the Virtex-6 FPGA ML605 Evaluation Kit. All operations, such as data processing, weight connections, multipliers, adders and activation function were performed using floating point format, because allows high precision in operations with high complexity. Some resources of Xilinx were used such as multipliers and CORE blocks, and the hyperbolic tangent of the activation is realized based on Taylor series. To validate the implementation results, we train a NARX network to model the inverse characteristics of a power amplifier. The results obtained in the simulation and the implementation shown a high correspondence.

Keywords: FPGA implementation, neural network, nonlinear behavior, Xilinx, floating point, Taylor series, CORDIC.

1 Introduction

Nowadays the Neural Networks (NN) should be used in problems that involve nonlinear behavior, and a high number of diverse NN topologies have been development for optimizing the performance in nonlinear modeling applications. In this context, the NARX network is one of the NN more precise for modeling nonlinear behavior. NARX network is a Recurrent Neural Network (RNN) with tapped delay lines in the input and output, which allows modeling the short and long-term dependencies [1]. However, at this moment, the NARX network only has been developing in simulations environment [2]. The hardware implementation of NARX network represents a new development.

Field Programmable Gate Array (FPGA) has been used for several implementations of NN with different mathematically models, applications and hardware characteristics [3-5]. FPGA provides some advantages, such as rapid prototyping, adaptation, reduced costs and simplicity in the design. The hardware implementation of the NN in FPGA involving several aspects must be taken into account, such as the data representation (fixed or floating point), the word length of the operands, the characteristics of the arithmetic operators (adder, multiplier, etc.), the activation function, the number of inputs and outputs of the network, and the structure of the network, (number of neurons, number of delay lines) [6].

This paper describes the FPGA implementation of the NARX network using Verilog language in Xilinx ISE Tool with the Virtex-6 FPGA ML605 Evaluation Kit[7]. To validate the implementation results, we train a NARX network to model the inverse characteristics of a power amplifier (PA).

2 NARX Neural Network

The NARX neural network is a RNN with tapped delay lines in the input and output vectors and its response is given by the discrete-time Nonlinear Autoregressive with eXogenous inputs (NARX) system [1]. The neural architecture of the NARX network is shown in Fig. 1. It architecture is formed by three mains layers: input, hidden, and output layer. The input layer is a set of tapped delay lines of the input and output signals, the hidden layer is a set of neurons with sigmoidal function activation, and the output layer is a set of neurons with linear activation function. Mathematically, the output in a NARX neural network is given by:

$$y(n) = \sum_{k=1}^{m} w_k^{LW} G_k(n) + b^2 \tag{1}$$

where

$$X_k(n) = \sum_{j=1}^{du} w_{(k,j)}^{IW} u(n-j) + \sum_{j=1}^{dy} w_{(k,j)}^{IW} y(n-j) + b_k^1 \quad (2)$$

and

$$G_k(n) = \tanh\left(X_k(n)\right) \quad (3)$$

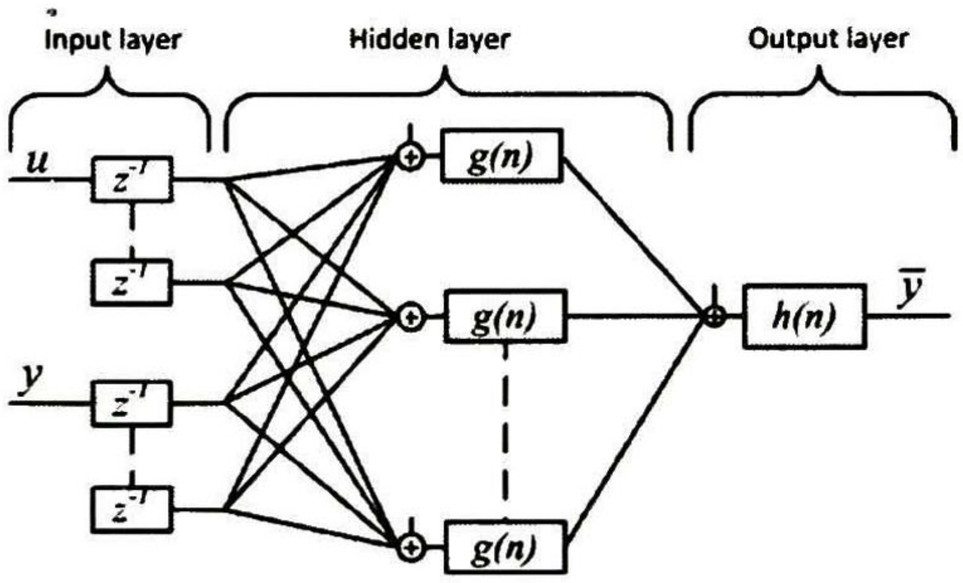


Fig. 1. Neural architecture of the NARX network.

The neuron model is shown in a Fig. 2.a. and the components in each neuron are: inputs (I), weights (w), bias (b), activation function (f), and adder (+). The scheme used in the implementation was realized based on the scheme presented in [8] and is shown in a Fig. 2.b. The basic operation that each neuron performs is initiated by the selection of the sample and its corresponds to 4 bit memory address of the weights matrix. Then, the multiplication of the connections between the inputs and weights are performed. The results are added and accumulated to all the connections. Next, the bias is added to each result of the sum and this value is passed through the activation function.

Floating point format is the data representation used, because allows high precision in operations with high complexity. The floating point format is commonly represented in simple precision of 32 bits and double precision of 64 bits.

The multiplier block is implemented in floating point with simple precision and its output is given by [9]

$$Z = (-1)^{s1 \oplus s2} (1.mant1 * 1.mant2) * 2^{e1+e2} \quad (4)$$

The sign is obtained with XOR operation.

The adder block is implemented in a CORE of Xilinx. Hyperbolic tangent is the activation function used. There are several methods and algorithms to obtain the hyperbolic tangent, such as LUTs, CORDIC [10] and hybrids methods. We proposed a method to find the hyperbolic tangent based on Taylor series. The hyperbolic tangent is given by

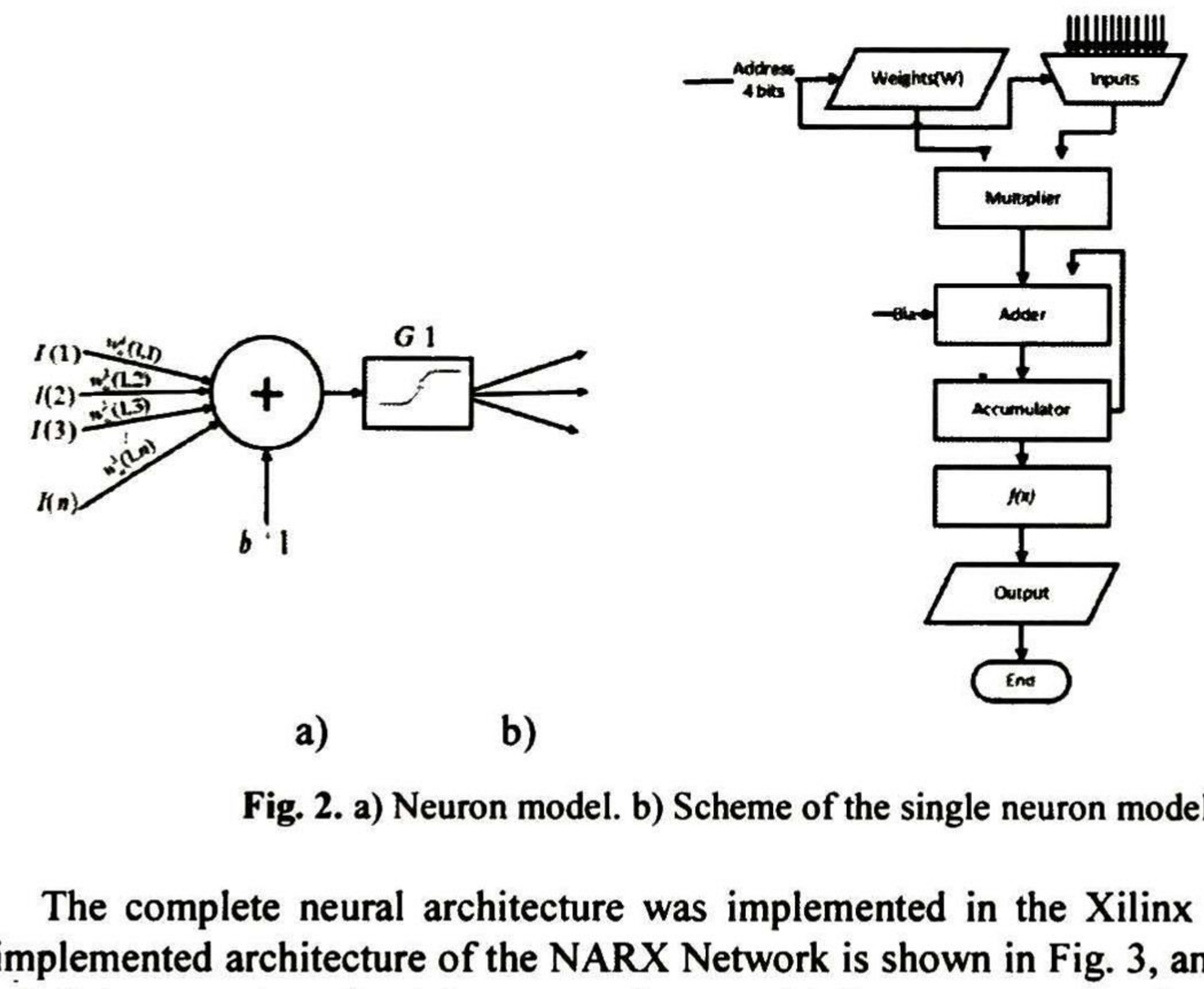
$$\tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}} \tag{5}$$

The expansion of the exponential by the Taylor series can be written as:

$$e^{x} = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \frac{x^5}{5!} + \ldots + \frac{x^n}{n!} \tag{6}$$

Then, using (5) in (6) and performing algebraic operations was obtained the equivalent function to represent the hyperbolic tangent, such is given by

$$\tanh(x) = \frac{362880x + 60480x^3 + 3024x^5 + 72x^7 + x^9}{362880 + 181440x^2 + 15120x^4 + 504x^6 + 9x^8} \tag{7}$$



a) b)

Fig. 2. a) Neuron model. b) Scheme of the single neuron model.

The complete neural architecture was implemented in the Xilinx ISE Tool. The implemented architecture of the NARX Network is shown in Fig. 3, and such consists of all the operation of weights connections, multipliers, sums, and activations functions.

Some resources of Xilinx were used, such as multipliers block, and CORE to realize the sums and divisions. The control of the neural architecture was realized by a Finite State Machine (FSM) type Mealy machine [11]. The output values are determined by both its current state and current inputs. The FSM has 20 states that perform 180 products, 192 sums, and 10 hyperbolic tangents, executed in 595 cycles.

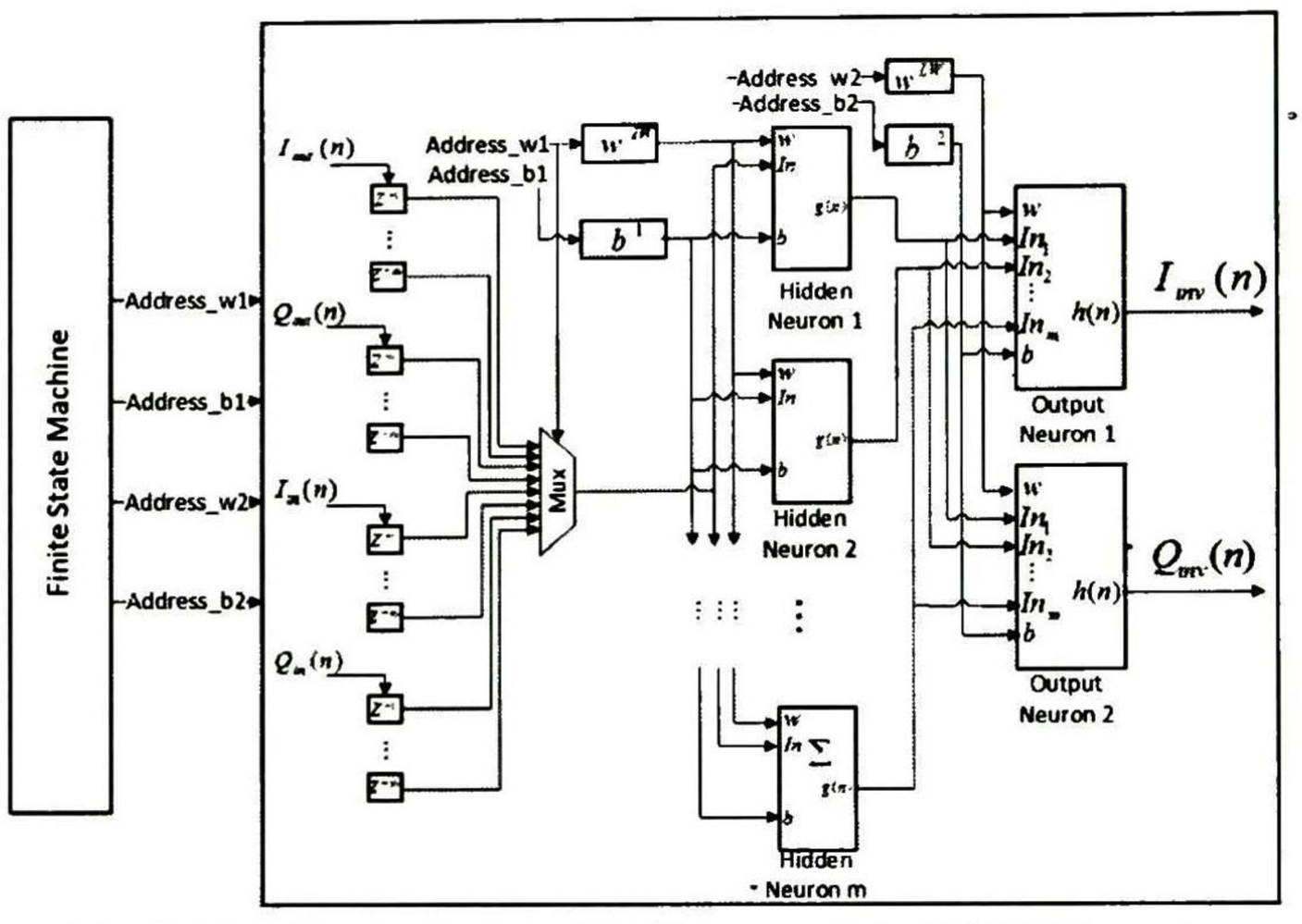


Fig. 3. Complete implemented architecture of the NARX Network.

2.1 Training and validation process

The training process was realized in MATLAB, and the weights matrices are stored in a memory unit. Prior to the training process, it is very important take into account the influence of several aspects, such as, the normalization [12], stop criterion [13], selected the memory order, number of neurons.

The normalization function is presented in Fig. 4. This processing is applied both in the input and output signals of the NARX network, and corresponding to the pre-processing and post-processing showed in Fig. 4.

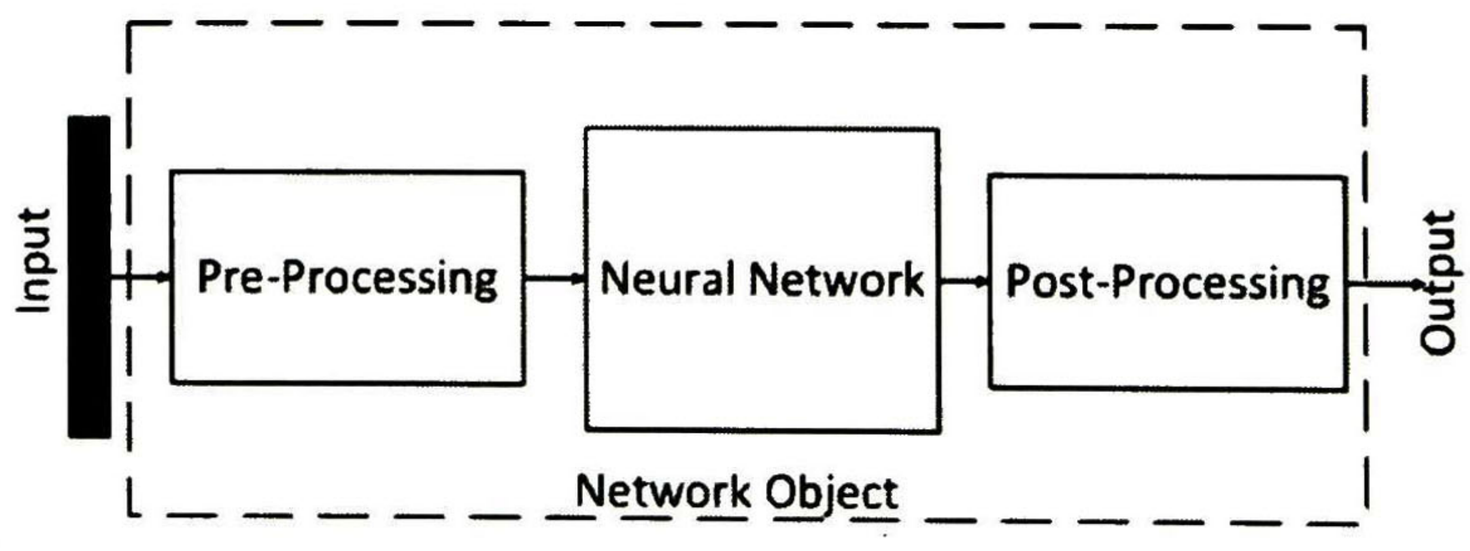


Fig. 4. Data processing in the input and output of the neural network.

and is given by:

$$y_{norm} = \frac{(y_{\max} - y_{\min}) * (x_{in} - x_{\min})}{(x_{\max} - x_{\min})} + y_{\min} \quad (1)$$

where y_{min} and y_{max} are the normalization values (+/-1), x_{min} and x_{max} are the maximum and minimum values of the vector and x_{in} is the actual data to process.

Final architecture of the NARX network implementation consists of pre-processing, processing, post-processing, communication and validation. Communication asynchrony (UART) [14] is realized to transfer the output data of the FPGA to the PC and validate the results, such as is shown in Fig. 5.

3 Experimentation and Results

An experimental setup to model the inverse characteristics of a PA is used to validate the FPGA implementation of the NARX network [2]. With the measurements of the input and output signals of the PA, the NARX network is training.

First, the parameters of the NARX network are defined. The input vector is formed by four inputs that represent the real values of the input and output signals of the PA. The number of delay lines selected was four for each one (input and output) and 10 neurons are selected in the hidden layer.

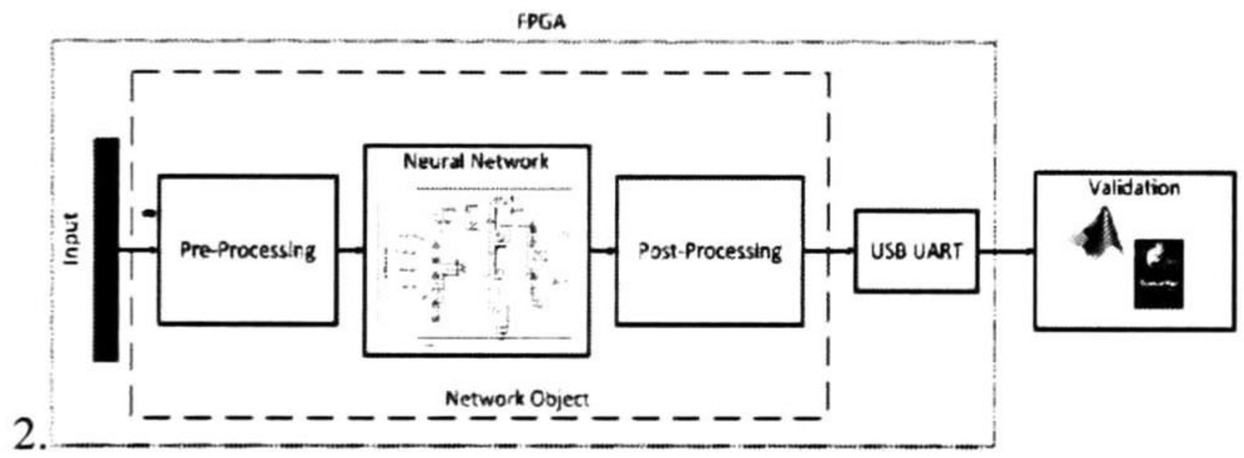


Fig. 1. Structure of the NARX network implementation.

Table 1. Comparison between the FPGA implementation and MATLAB simulation results.

Results	Real(FPGA)	Image(FPGA)	Real(Matlab)	Image(Matlab)
1	-0.34510845317838	0.07818963943093	-0.34510342853533	0.07818700472594
2	-0.35257565491600	-0.02967927526307	-0.35257891077726	-0.02967838693830
3	-0.24657515530991	-0.07807854950230	-0.24664802091774	-0.07809724509940
4	-0.06395308502237	-0.03039482902367	-0.06402700042977	-0.03042185011335
5	0.12606102178778	0.07410325482868	0.12605209565504	0.07410446812629
6	0.25096877400903	0.17675751911770	0.25096190357323	0.17675773608755
7	0.28196600764153	0.20763142923692	0.28196250978712	0.20762731818684
8	0.21321615559612	0.15994817431051	0.21321632703128	0.15994433789154
9	0.12060241480182	0.06695307107320	0.12060398021669	0.06695367249790
10	0.03242863820595	0.00092798982272	0.03242695709790	0.00092780150537

The neural architecture of the NARX network was training in MATLAB and the weights and biases matrices are store in a RAM block. Then, the neural architecture training was implemented in a FPGA Virtex-6 and validated its response with the same signals used in the training.

The results obtained after the communication between the FPGA and the PC are compared with the results of the simulation obtained in MATLAB, in order to demonstrate the validity of the implementation. Experimental results are shown in Table 1 and shown a high correlation between MATLAB and FPGA.

It can be seen that the differences between the implementation and simulation results only are present after the six decimal, and this is produced because the implementation in the FPGA is based on 32 bits and the simulation in MATLAB used 64 bits. The error calculated is less than 1%. Table 2 gives the resource requirements reported by Xilinx in details.

Table 1. Resource utilization of the Virtex-6

Resource utilization	Hidden Neuron	Output Neuron	Multiplier	Hyperbolic Function	Adder	Total Used	Available
Slice Registers	3699	672	97	3027	578	38572	301440
Slice LUTs	2697	573	96	2131	446	29057	150720
LUT FF	2116	461	79	1645	354	21491	89013
Bonded IOBs	137	135	100	68	68	3	600
Block RAM						225	416
DSP	6	2	2	4	2	64	768
Maximum Operation Frequency	-			95.511 MHz			

4 Conclusions

The implementation of a NARX neural network in a FPGA has been presented, showing a high correlation between the data obtained with the FPGA and the simulation in MATLAB. An error is presented in the less significant bits. Also it is presented a method to implement the exponential function by means of an expansion of the Taylor series, that can be more complex but with higher accuracy with respect to others methods. The NARX was implemented in a sequential form, a disadvantage with architectures as pipeline or segmented. The application used to validate of the implementation was verified with the modeled of the inverse characteristics of a PA. The validation results in the modeled of nonlinear behavior shown that the FPGA implementation of the NARX network are efficient.

5 References

1. H. T. Siefelmann, B. G. Horne and C. L. Giles, "Computational capabilities of recurrent NARX neural networks," *IEEE Transactions on Systems, Man and Cybernetics-Part B: Cybernetics,* Vol. 27, No.2, pp. 208-215, Apr. 1997.
2. L.M. Aguilar-Lobo, A. Garcia-Osoria, J.R. Loo-Yau, S. Ortega-Cisneros, P. Moreno, J.E. Rayas-Sanchez, J.A. Reynoso-Hernández, "A Digital Predistortion Technique Based on a NARX Network to Linearize GaN Class F Power Amplifiers", *IEEE* 57th International Midwest Symposium on Circuits And Systems, August 2014.
3. M. Bahoura, C.-W. Park, "FPGA-Implementation of an Adaptive Neural Network for RF Power Amplifier Modeling", in *New Circuits and Systems Conference (Newcas). 2011 IEEE 9th International, june 2011, pp.29-32.*
4. M. Atencia, H. Boumeridja, G. Joya, F. Garcia-Lagos and F.Sandoval, "FPGA Implementation of a Systems Identification Module Based Upon Hopfield Networks", *Neurocomputing,* 70(2007) 2828-2835, 2007.
5. J.L. Bastos, H.P. Figueroa, A. Monti, "FPGA Implementation of Neural Networks-Based Controllers for Power Electronics Applications", *in Applied Power Electronics Conference and Exposition, 2006. APEC '06. Twenty-First Annual IEEE,* 2006, pp. 1–6.
6. A.L.S. Braga, C.H. Llanos, D. Gohringer, J. Obie, J. Becker, M. Hubner, "Performance, Accuracy, Power Consumption and Resource Utilization Analysis for Hardware/Software realized Artificial Neural Networks", *Bio-inspired Computing: Theories and Aplications (BIC-TA), 2010 IEEE Fifth Internatiol Conference on,* pp.1629-1636. Sept.2010.
7. "Virtex-6 FPGA ML605 Evaluation Kit", www.xilinx.com/products/boards-and-kits/EK-V6-ML-605-G.htm, Xilinx Inc.
8. K. Mohamad, M. F. O. Mahmud, F. H. Adnan, W. F. H. Abdullah. "Design of single neuron on FPGA", Humanities, Science and Engineering Research (SHUSER) 2012 IEEE Symposium on.
9. IEEE Standard for Floating-Point Arithmetic (IEEE Std 754-2008), Revision of IEEE Std 754-4985. 29 August 2008.
10. Meng Qian, "Application of CORDIC Algorithm to Neural Networks VLSI Design", *in Multconf. Computational Engineering in Systems Applications IMACS.* Oct. 2006, pp.504-508.
11. Z.Salcic, A. Smailagic, "Digital system design and prototyping using field programmable logic", Boston: Kluwer Academic Publishers, 1997, pp. 134-141.
12. M. A. Sartin, A. C. R. da Silva "Approximation of Hyperbolic Tangent Activation Function Using Hybrid Methods", Department of Computing, UNEMAT- Universidade do Estado de Mato Grosso, Colider, MT, Brazil.
13. W. Wang, P. H. A. J. M. Van Gelder and J. K. Vrijling, "Some issues about the generalization of neural networks for time series prediction"
14. *ICANN 2005*, LNCS 3697, pp. 559-564, 2005.G. B. Wakhle, I. Aggarwal, S. Gaba, "Synthesis and Implementation of UART Using VHDL Codes", *International Symposium on Computer, Consumer and Control*, 2012.

"Neural Network Toolbox, User's Guide R2014a", www.mathworks.com/help/pdf_doc/nnet/nnet_ug.pdf, The MathWorks.Inc, Pg.2-9, 2-10, 2-11.

Apéndice A

En este apéndice se muestra todo el código fuente desarrollado a lo largo del proyecto, se incluyen el código implementado en Verilog y los scripts implementados en Matlab para la recepción de datos de la comunicación serial y la generación de los archivos de pesos y polarizaciones de la red neuronal.

A.1 Código Fuente

A.1.1 Código fuente para el modulo Top-Level

El código fuente mostrada a continuación, es el bloque principal que se encargado de realizar el procesamiento de la información y comunicación serial.

```
//Descripcion del modulo y puertos de entrada y salida
//-----------------------------------------
module Trans_TX(
input  clk,
input  start,
output ser );

//Declaracion de los registros y conexiones
//-----------------------------------------
reg reset=0;
reg [17:0] address=0;
wire rdy, rdy1 ,ready;
wire [31:0] Imag, Real;
wire [63:0] doble;
reg [31:0] data;
reg mux,In_R, In_NX;
reg [2:0] std;
reg pls, scl_flanco;

//monitoreo del boton de inicio
//-------------------------------
```

```
always @(posedge clk)
begin
if(start)
pls<=1;
else
pls<=0;
end

always@(*)
begin
scl_flanco=start&~pls;
end

// Instanciar moduls contenidos en el top-level
//-----------------------------------------------------------------
RED_NARX   inst1(.clk(clk),   .start(In_NX),   .reset(reset),   .Act_Imag(Imag),
.Act_Real(Real), .ready_out(rdy), .dir_address(address));

Single_Double inst2(.clk(clk), .sample(rdy|In_R), .reset(reset), .simple(data),
.ready(rdy1), .double(doble));

Serial   inst3(.clk(clk),      .start(rdy1),      .entrada(doble),      .ser(ser)
,.ready(ready));

//FSM empleada para controlar el flujo de datos de las instancias
//-----------------------------------------------------------------
always @(posedge clk)
begin
case(std)
0:begin if(scl_flanco) std<=1; else begin  std<=0; mux<=0; In_R<=0; In_NX<=0;
address<=0; end end
1:begin In_NX<=0; mux<=0; In_R<=0; std<=2; end
2:begin In_NX<=1; mux<=0; In_R<=0; std<=3; end
3:begin In_NX<=0; if(ready) begin mux<=1; std<=4; end else begin mux<=0; std<=3;
end end
4:begin In_R<=1; std<=5;end
5:begin In_R<=0; if(ready)   std<=6;  else  std<=5; end
6:begin if(address==61435) std<=0; else begin address<=address+1; std<=7; end end
7:begin std<=1;end
endcase
end

//multiplexado de dato se transmision serial
//-------------------------------------
always @(*)
begin
if(mux)
data=Real;
else
data=Imag;
end
//-------------------------------------
endmodule
```

A.1.2 Código Fuente de la red NARX

El siguiente código fuente es la implementación de la red neurona NARX, y la máquina de estados empleada para el control y flujo de datos.

```
//Declaracion del modulo principal y las entradas necesarias
//-----------------------------------------------------------------
module RED_NARX(
input clk,
input reset,
input start,
input [17:0] dir_address,
```

```
output reg ready_out,
output reg [31:0] Act_Imag=0, Act_Real=0);

//Declaracion de conexiones de la Red y registros necesarios
//----------------------------------------------------------
wire [9:0] ready, ready2;
wire [31:0] w11, bias, bias2, w12, w13, w14, w15, w16, w17, w18, w19, w110, w21,
w22;
wire [31:0] Act1, Act2, Act3, Act4, Act5, Act6, Act7, Act8, Act9, Act10, Imag,
Real;
wire [31:0] IN;
wire rdy, rdy1;
reg sample, RW, b1, sample2, b2;
reg [9:0] act_tng;
reg [3:0] address,address_bias, address2;
reg r_sum,.r_sal;
reg [9:0] add_bia;
reg [4:0] std=0;
reg [9:0] finish;
reg [31:0] capa_out;
reg [1:0] add_bia2;
reg address_bias2;
reg [17:0] address_In;

//FSM para el control de la red NARX
//----------------------------------------------
always @(posedge clk or posedge reset)
begin
if(reset)
std<=0;
else
begin
case(std)
0:begin    if(start)  std<=1;  else  begin  ready_out<=0;  sample2<=0;  r_sal<=0;
add_bia2<=0; address_bias2<=0; b2<=0; RW<=0; std<=0; address<=0; address_bias<=0;
add_bia<=0; act_tng<=0; sample<=0;  r_sum<=0;  b1<=0; finish<=0; address2<=0;end
end
1:begin  sample<=0; std<=2; finish<=0; end
2:begin sample<=0; std<=3; end
3:begin  sample<=0; std<=4;  end
4:begin  sample<=1; std<=5; end
5:begin    if(finish==10'h3ff)    std<=6;    else    begin    std<=5;    sample<=0;
finish<=finish|ready; end end
6:begin if(address==4'hf) begin std<=7; address<=0; b1<=1; finish<=0; end  else
begin address<=address+1; std<=1; end end

7:begin std<=8; add_bia[address_bias]<=1; end
8:begin std<=9;  add_bia<=add_bia<<0; end
9:begin  act_tng<=ready;  add_bia<=0;  if(ready2>0)begin  finish<=finish|ready2;
std<=10; if(address==4)address_bias<=0; else address_bias<=address_bias+1;  end
else begin std<=9; end end
10:begin if(finish==10'h3ff) begin std<=11; r_sum<=1;  end else std<=7; end

11:begin sample2<=0; std<=12; b1<=0; r_sum<=0; finish<=0; end
12:begin sample2<=1; std<=13; end
13:begin sample2<=0; std<=14; end
14:begin if(rdy) std<=15; else std<=14; end
15:begin  if(address2==4'h9)  begin  std<=16;  b2<=1;  address<=0;  end  else  begin
address2<=address2+1; std<=11;end end
16:begin std<=17; add_bia2<=1; end
17:begin std<=18; add_bia2<=0; end
18:begin  if(rdy)  begin   Act_Imag<=Imag;  std<=19;  address_bias2<=1;  end  else
std<=18; end
19:begin std<=20; add_bia2<=0; end
20:begin std<=21; add_bia2<=2; end
21:begin std<=22; add_bia2<=0; end
22:begin if(rdy1) begin  Act_Real<=Real; std<=23; end else std<=22; end
23:begin std<=0; b2<=0; r_sal<=1; ready_out<=1; end
default:std<=0;
endcase
end
```

```
end

//Selecciones de direcciones de memoria de datos de entrada, permite seleccinas
//de acuerdo al algoritmo para selección de retardos
//-------------------------------------------------
always @(posedge clk)
begin
case(address)
  0:  address_In<= 3 +dir_address;
  1:  address_In<=61443 +dir_address;
  2:  address_In<=2 +dir_address;
  3:  address_In<=61442 +dir_address;
  4:  address_In<=1+dir_address;
  5:  address_In<=61441+dir_address;
  6:  address_In<=dir_address;
  7:  address_In<=61440+dir_address;
  8:  address_In<=122883+dir_address;
  9:  address_In<=184323+dir_address;
  10: address_In<=122882+dir_address;
  11: address_In<=184322+dir_address;
  12: address_In<=122881+dir_address;
  13: address_In<=184321+dir_address;
  14: address_In<=122880+dir_address;
  15: address_In<=184320+dir_address;

  default: address_In<=3;
endcase
end

//Seleccion de dato a procesar en la capa de salida
//-------------------------------------------------
always @(*)
begin
case(address2)
   0: capa_out=Act1;
   1: capa_out=Act2;
   2: capa_out=Act3;
   3: capa_out=Act4;
   4: capa_out=Act5;
   5: capa_out=Act6;
   6: capa_out=Act7;
   7: capa_out=Act8;
   8: capa_out=Act9;
   9: capa_out=Act10;

  default: capa_out=Act1;
endcase
end

//Declaracion de Instancias de unidades de almacenamiento
//-------------------------------------------------------

Mem_In inst1      (.clka(clk), .wea(RW), .addra(address_In), .dina(32'h00000000),
.douta(IN));

Mem_W1 inst2(.clka(clk),    .wea(RW),    .addra(address),    .dina(32'h00000000),
.douta(w11));

Mem_W2 inst3(.clka(clk),    .wea(RW),    .addra(address),    .dina(32'h00000000),
.douta(w12));

Mem_W3 inst4(.clka(clk),    .wea(RW),    .addra(address),    .dina(32'h00000000),
.douta(w13));

Mem_W4 inst5(.clka(clk),    .wea(RW),    .addra(address),    .dina(32'h00000000),
.douta(w14));

Mem_W5 inst6(.clka(clk),    .wea(RW),    .addra(address),    .dina(32'h00000000),
.douta(w15));
```

```
Mem_W6 inst7(.clka(clk),     .wea(RW),     .addra(address),     .dina(32'h00000000),
.douta(w16));

Mem_W7 inst8(.clka(clk),     .wea(RW),     .addra(address),     .dina(32'h00000000),
.douta(w17));

Mem_W8 inst9(.clka(clk),     .wea(RW),     .addra(address),     .dina(32'h00000000),
.douta(w18));

Mem_W9 inst10(.clka(clk),     .wea(RW),     .addra(address),     .dina(32'h00000000),
.douta(w19));

Mem_W10 inst11(.clka(clk),     .wea(RW),     .addra(address),     .dina(32'h00000000),
.douta(w110));

Mem_W21 inst12(.clka(clk),     .wea(RW),     .addra(address2),     .dina(32'h00000000),
.douta(w21));

Mem_W22 inst15(.clka(clk),     .wea(RW),     .addra(address2),     .dina(32'h00000000),
.douta(w22));

Mem_B2 inst13(.clka(clk),  .wea(RW),  .addra(address_bias2),  .dina(32'h00000000),
.douta(bias2));

Mem_B1    inst14(.clka(clk),  .wea(RW),  .addra(address_bias),  .dina(32'h00000000),
.douta(bias));

// alambrado de capa oculta capa oculta
//-------------------------------------------------------------------------------
NARX  inst18     (.clk(clk),  .reset(reset),  .start(sample),  .IN(IN),  .W(w11),
.bias(bias),     .reset_sum(r_sum),     .Activacion(Act1),     .add_bia(add_bia[0]),
.sample_act(act_tng[0]), .rdy_sum(ready[0]), .rdy_act(ready2[0]), .b1(b1));

NARX  inst19     (.clk(clk),  .reset(reset),  .start(sample),  .IN(IN),  .W(w12),
.bias(bias),     .reset_sum(r_sum),     .Activacion(Act2),     .add_bia(add_bia[1]),
.sample_act(act_tng[1]), .rdy_sum(ready[1]), .rdy_act(ready2[1]), .b1(b1));

NARX  inst20     (.clk(clk),  .reset(reset),  .start(sample),  .IN(IN),  .W(w13),
.bias(bias),     .reset_sum(r_sum),     .Activacion(Act3),     .add_bia(add_bia[2]),
.sample_act(act_tng[2]), .rdy_sum(ready[2]), .rdy_act(ready2[2]), .b1(b1));

NARX  inst21     (.clk(clk),  .reset(reset),  .start(sample),  .IN(IN),  .W(w14),
.bias(bias),     .reset_sum(r_sum),     .Activacion(Act4),     .add_bia(add_bia[3]),
.sample_act(act_tng[3]), .rdy_sum(ready[3]), .rdy_act(ready2[3]), .b1(b1));

NARX  inst22     (.clk(clk),  .reset(reset),  .start(sample),  .IN(IN),  .W(w15),
.bias(bias),     .reset_sum(r_sum),     .Activacion(Act5),     .add_bia(add_bia[4]),
.sample_act(act_tng[4]), .rdy_sum(ready[4]), .rdy_act(ready2[4]), .b1(b1));

NARX  inst23     (.clk(clk),  .reset(reset),  .start(sample),  .IN(IN),  .W(w16),
.bias(bias),     .reset_sum(r_sum),     .Activacion(Act6),     .add_bia(add_bia[5]),
.sample_act(act_tng[5]), .rdy_sum(ready[5]), .rdy_act(ready2[5]), .b1(b1));

NARX  inst24     (.clk(clk),  .reset(reset),  .start(sample),  .IN(IN),  .W(w17),
.bias(bias),     .reset_sum(r_sum),     .Activacion(Act7),     .add_bia(add_bia[6]),
.sample_act(act_tng[6]), .rdy_sum(ready[6]), .rdy_act(ready2[6]), .b1(b1));

NARX  inst25     (.clk(clk),  .reset(reset),  .start(sample),  .IN(IN),  .W(w18),
.bias(bias),     .reset_sum(r_sum),     .Activacion(Act8),     .add_bia(add_bia[7]),
.sample_act(act_tng[7]), .rdy_sum(ready[7]), .rdy_act(ready2[7]), .b1(b1));

NARX  inst26     (.clk(clk),  .reset(reset),  .start(sample),  .IN(IN),  .W(w19),
.bias(bias),     .reset_sum(r_sum),     .Activacion(Act9),     .add_bia(add_bia[8]),
.sample_act(act_tng[8]), .rdy_sum(ready[8]), .rdy_act(ready2[8]), .b1(b1));

NARX  inst27     (.clk(clk),  .reset(reset),  .start(sample),  .IN(IN),  .W(w110),
.bias(bias),     .reset_sum(r_sum),     .Activacion(Act10),     .add_bia(add_bia[9]),
.sample_act(act_tng[9]), .rdy_sum(ready[9]), .rdy_act(ready2[9]), .b1(b1));

//Alambrado de capa de salida
//-------------------------------------------------------------------------------
```

```
NARX_OUT  inst31    (.clk(clk),  .reset(reset),  .start(sample2),  .IN(capa_out),
.W(w21),  .bias(bias2),  .reset_sal(r_sal),.result(Imag),  .add_bia(add_bia2[0]),
.rdy_out(rdy), .b1(b2));

NARX_OUT  inst32    (.clk(clk),  .reset(reset),  .start(sample2),  .IN(capa_out),
.W(w22),  .bias(bias2),  .reset_sal(r_sal),.result(Real),  .add_bia(add_bia2[1]),
.rdy_out(rdy1), .b1(b2));

endmodule
```

A.1.3 Implementación de Neurona

A continuación se muestra el código fuente para una neurona de múltiples entradas.

```
// declaracion del modulo entradas y salidas
//----------------------------------------
module NARX(
input clk,
input reset,
input reset_sum,
input b1,
input start,
input [31:0] IN,
input [31:0] W, bias,
input add_bia, sample_act,
output [31:0] Activacion,
output rdy_sum, rdy_act);

// Declaracion de conexiones y registros
//----------------------------------------
wire [31:0] result_tmp;
wire rdy1;
reg [31:0] net;
wire [31:0] result;

//instanciacion de modulos
//------------------------------------------------
Float_Mult_AB  inst1(.clk(clk),  .reset(reset),  .sample(start),  .data_a(IN),
.data_b(W), .ready(rdy1), .result(result_tmp));

Sumatoria inst2(.clk(clk), .data_a(net), .sample(rdy1|add_bia), .ready(rdy_sum),
.acumulador(result), .reset(reset_sum));

Tanh  inst3(.clk(clk),  .sample(sample_act),  .reset(reset),  .phase(result),
.ready(rdy_act), .tangente(Activacion));

//multiplexor para seleccion de peso o polariacion
//------------------------------------------------
always@(rdy1 or b1)
begin
  if(b1)
    net=bias;
  else
    net=result_tmp;
end

endmodule
```

A.1.4 Multiplicación en coma flotante

```
module Float_Mult_AB(clk, reset, sample, data_a, data_b, ready, result);
input clk;
```

```
input reset;
input sample;
input [31:0] data_a;
input [31:0] data_b;
output reg ready=0;
output reg [31:0] result=0;

reg [2:0] std=0;
reg [7:0] exp_tmp=0;
reg [47:0] mantisa_tmp=0;
reg [4:0] Q=0;

always @(posedge clk)
begin
  if(reset)
  std<=0;
else
  begin
   case(std)
     0: begin if (sample) std<=1; else std<=0; end
     1: begin
          exp_tmp<=data_a[30:23]+data_b[30:23]-127;
          mantisa_tmp<={1'b1,data_a[22:0]}*{1'b1,data_b[22:0]};
          std<=2;
        end
     2: begin
        if(mantisa_tmp[47])
           std<=4;
        else
           begin
             std<=3;
             mantisa_tmp<=mantisa_tmp<<1;
             Q<=Q+1;
           end
        end
  3: begin
       if(mantisa_tmp[47])
        std<=4;
      else
       begin
         std<=2;
         mantisa_tmp<=mantisa_tmp<<1;
         Q<=Q+1;
       end
     end
  4: begin
        result[31]<=data_a[31]^data_b[31];
        result[30:23]<=exp_tmp-Q+1;
        result[22:0]<= mantisa_tmp[46:24];
        ready<=1;
        std<=5;
      end
 5:begin
       exp_tmp<=0;
       mantisa_tmp<=0;
             Q<=0;
             ready<=0;
             std<=0;
      end
     default: std<=0;
    endcase
 end
end
endmodule
```

A.1.5 Sumatoria en coma flotante

```
module Sumatoria(
input clk,
input [31:0] data_a,
input sample,
```

```
input reset,
output ready,
output reg [31:0] acumulador=0
    );

wire [31:0] result;
wire rdy;

//Instanciacion de Core Flotating-Point
//-----------------------------------------------
Sum_Float   inst1(.clk(clk),   .operation_nd(sample),   .rdy(ready),   .a(data_a),
.b(acumulador), .result(result));
//-----------------------------------------------
always @(*)
  begin
    if(reset)
        acumulador=0;
    else if(ready)
      begin
        acumulador=result;
      end
  end
endmodule
```

A.1.6 Tangente Hiperbólica

```
module Tanh(
input clk,
input reset,
input sample,
input [31:0] phase,
output reg ready,
output reg [31:0] tangente=0
    );

wire rdy, rdy_num, rdy_den, rdy_out;
wire [31:0] X1,X2,X3,X4,X5,X6,X7,X8,X9,tang_tmp;
wire [31:0] numerador, denominador;
reg [31:0] net1, net2;

reg [2:0]  sum;
reg [3:0] std;
reg add, div, r_sum, init;
reg [1:0] salida=0;

Taylor inst1(.clk(clk),        .reset(reset),        .sample(init),.phase(phase),
.ready(rdy),.X1(X1),.X2(X2),.X3(X3),.X4(X4),                          .X5(X5),
.X6(X6),.X7(X7),.X8(X8),.X9(X9));

Sumatoria   inst2(.clk(clk),   .reset(r_sum),   .sample(add),   .data_a(net1),
.ready(rdy_num), .acumulador(numerador));

Sumatoria   inst3(.clk(clk),   .reset(r_sum),   .sample(add),   .data_a(net2),
.ready(rdy_den), .acumulador(denominador));

Div_Float  inst4(.clk(clk),  .operation_nd(div),  .a(numerador),  .b(denominador),
.rdy(rdy_out), .result(tang_tmp));

always @(posedge clk)
 begin
   case (salida)
     0: tangente=tangente;
     1:tangente=32'h3f800000;
     2: tangente=32'hbf800000;
     3:tangente= tang_tmp;
    default: tangente=tangente;
  endcase
end
```

```
always@(posedge clk)
begin
if(reset)
std<=0;
else
begin
case(std)
0:begin if(sample) std<=1; else begin add<=0; std<=0; div<=0; sum<=0; r_sum<=0;
init<=0; ready<=0; salida<=0; end end
1:begin if(phase[30:0]>31'h40551eb8) std<=2; else std<=3; end
2:begin std<=10; if(phase[31]==1)salida<=2; else salida<=1; end
3:begin init<=1; std<=4; end
4:begin init<=0; if(rdy) std<=5; else std<=4; end
5:begin sum<=0; std<=6; end
6:begin add<=1; std<=7;end
7:begin if(rdy_num) std<=8; else begin add<=0; std<=7; end end
8:begin if(sum==4) begin div<=1; std<=9; end else begin sum<=sum+1; std<=6; end
end
9:begin add<=0; div<=0; if(rdy_out)begin std<=10; salida<=3; end else std<=9; end
10:begin std<=0; r_sum<=1; ready<=1; end
default:std<=0;
endcase
end
end

//multiplexores de datos

always @(*)
begin
   case(sum)
     0:net1=X1;
     1:net1=X3;
     2:net1=X5;
     3:net1=X7;
     4:net1=X9;
    default:net1=X1;
   endcase
end

always @(*)
begin
   case(sum)
     0:net2=32'h48b13000;
     1:net2=X2;
     2:net2=X4;
     3:net2=X6;
     4:net2=X8;
    default:net2=32'h48b13000;
  endcase
end
endmodule
```

A.1.7 Calculo de los polinomios de Taylor

```
module Taylor(
input clk,
input reset,
input sample,
input [31:0] phase,
output reg ready=0,
output reg [31:0] X1=0,X2=0,X3=0, X6=0,X4=0,X5=0,X7=0,X8=0,X9=0

    );


reg [3:0] std=0;
reg [3:0] pot=0;
```

```
reg muestra;
reg [31:0] acumulador, entrada=0;

wire [31:0] result, result1;
wire rdy, rdy1;


always @(posedge clk)
begin
if(reset)
std<=0;
else
begin
case(std)
0:begin if(sample) std<=1; else begin std<=0; muestra<=0; ready<=0; end end
1:begin pot<=0; std<=2; end
2:begin muestra<=1; ready<=0; std<=3; end
3:begin if(rdy1)begin std<=4; end else begin muestra<=0; std<=3; end end
4:begin if(pot==8)std<=5; else begin pot<=pot+1; std<=2;end end
5:begin muestra<=0; std<=6; ready<=1; end
6:begin std<=0; ready<=0;end
default:std<=0;
endcase
end
end

always @(posedge clk)
begin
     if(pot==0)
       acumulador=32'h3f800000;
    else if(std==1)
      acumulador=phase;
    else
      acumulador=result;
end


always @(posedge clk)
begin
    if(rdy1)
     begin
      case(pot)
       0: X1=result1;
       1: X2=result1;
       2: X3=result1;
       3: X4=result1;
       4: X5=result1;
       5: X6=result1;
       6: X7=result1;
       7: X8=result1;
       8: X9=result1;
      default:X9=X9;
    endcase
  end
end


always @(posedge clk)
begin
   if(rdy)
    begin
     case(pot)
       0: entrada=32'h48b13000;//362880
       1: entrada=32'h48313000;//181440
     2: entrada=32'h476c4000;//60480
     3: entrada=32'h466c4000;//15120
     4: entrada=32'h453d0000;//3024
     5: entrada=32'h43fc0000;//504
     6: entrada=32'h42900000;//72
     7: entrada=32'h41100000;//9
     8: entrada=32'h3f800000;//1
   default: entrada=32'h48313000;
```

```
  endcase
end
end

Float_Mult_AB inst1(.clk(clk), .reset(reset), .sample(muestra), .data_a(phase),
.data_b(acumulador), .ready(rdy), .result(result));

Float_Mult_AB  inst2(.clk(clk),  .reset(reset),  .sample(rdy),  .data_a(entrada),
.data_b(result), .ready(rdy1), .result(result1));

endmodule
```

A.1.8 Neurona de la capa de salida

```
module NARX_OUT(
input clk,
input reset,
input reset_sal,
input b1,
input start,
input [31:0] IN,
input [31:0] W, bias,
input add_bia,
output [31:0] result,
output rdy_out
    );

wire [31:0] result_tmp;
wire rdy;
reg [31:0] net;


Float_Mult_AB  inst1(.clk(clk),  .reset(reset),  .sample(start),  .data_a(IN),
.data_b(W), .ready(rdy), .result(result_tmp));

Sumatoria inst2(.clk(clk), .reset(reset_sal), .data_a(net), .sample(rdy|add_bia),
.ready(rdy_out), .acumulador(result));

always@(rdy or b1)
begin
    if(b1)
     net=bias;
    else
     net=result_tmp;
end
endmodule
```

A.1.9 Modulo para convertir de precisión simple a doble

```
module Single_Double(
input clk,
input reset,
input sample,
input [31:0] simple,
output  reg ready,
output reg [63:0] double=0
    );

reg [2:0] std=0, next_std=0;
reg [10:0] exp_tmp=0;

always @(posedge clk)
begin
    if (reset)
      std<=0;
    if(sample)
      std<=1;
```

```
    else
      std<=next_std;
end

always @(*)
begin
   case(std)
    0: next_std=0;
    1: next_std=2;
    2:next_std=3;
    3: next_std=0;
   default: next_std=std;
endcase
end

always@(posedge clk)
begin
   case(std)
    0: begin ready<=0;  end
    1:exp_tmp<=simple[30:23]+896;
    2:begin
        double[63]<=simple[31];
        double[62:52]<=exp_tmp;
        double[51:0]<={simple[22:0],29'h00000000};
        ready<=1;
      end
   3:begin
        exp_tmp<=0;
        ready<=0;
      end
 endcase
end
endmodule
```

A.1.10 Transmisión RS-232

```
module Serial(input clk, input [63:0] entrada,  output ser, input start, output
reg ready);


// Start signal tells it to start sending bits
//reg start;

//The bits of data to send

parameter baudios=258; // para 115200 Bd--573 ciclos, 256000 bd--258 ciclos

wire [7:0] data_tx;
reg init=0;

reg [4:0] B_tx,ent;

reg RST;

Decoder inst1(.data(ent),.dec(data_tx));

always @(*)
begin
case(B_tx)
0: ent=entrada[63:60];
1: ent=entrada[59:56];
2: ent=entrada[55:52];
3: ent=entrada[51:48];
4: ent=entrada[47:44];
5: ent=entrada[43:40];
6: ent=entrada[39:36];
7: ent=entrada[35:32];
8: ent=entrada[31:28];
```

```
9: ent=entrada[27:24];
10: ent=entrada[23:20];
11: ent=entrada[19:16];
12: ent=entrada[15:12];
13: ent=entrada[11:8];
14: ent=entrada[7:4];
15: ent=entrada[3:0];
16: ent=16;
default: ent=entrada[63:60];
endcase
end

//////////////////////divisor de frecuencia para transmision

reg [12:0] clockdiv=0;
always @(posedge clk)
begin
    if (clockdiv == baudios)
        clockdiv <= 0;
 else if(init)
clockdiv<=1;
    else
        clockdiv <= clockdiv + 1;
end

// The serclock is a short pulse each time we are reset
wire serclock = (clockdiv == 0);


//////////////////////maquina de control

reg [3:0] state;

always @(posedge clk)
begin
   case (state)
  0: begin if(start) begin  B_tx<=0; state<=1; end else state<=0; ready<=0; end
  1: begin RST<=1; state<=2; end
       2: begin if (RST) begin state <= 3; init<=1; end else state<=2; end
  3: begin init<=0; state<=4; end
       4: if (serclock) state <= 5;     // Start bit
       5: if (serclock) state <= 6;     // Bit 0
       6: if (serclock) state <= 7;     // Bit 1
       7: if (serclock) state <= 8;     // Bit 2
       8: if (serclock) state <= 9;     // Bit 3
       9: if (serclock) state <= 10;    // Bit 4
       10: if (serclock) state <= 11;   // Bit 5
       11: if (serclock) state <= 12;   // Bit 6
       12: if (serclock) state <= 13;   // Bit 7
  13: begin if (serclock) begin if(B_tx==16) state<=15; else begin state <= 14;
B_tx<=B_tx+1;end end end
  14: if (serclock) state <= 5;
       15: if (serclock) begin state <= 0;  RST<=0; ready<=1; end // Stop bit
       default: state <= 0;                  // Undefined, skip to stop
    endcase
end


//////////////////////salida del dato serial
reg outbit;

always @(posedge clk)
begin
    case (state)
         0: outbit <= 1;                 // idle
1: outbit <= 1;    // idle
2: outbit <= 1;    //idle
3: outbit <= 1;    //idle
         4: outbit <= 0;                 // Start bit
         5: outbit <= data_tx[0];          // Bit 0
         6: outbit <= data_tx[1];          // Bit 1
```

```
            7: outbit <= data_tx[2];          // Bit 2
            8: outbit <= data_tx[3];          // Bit 3
            9: outbit <= data_tx[4];          // Bit 4
            10: outbit <= data_tx[5];          // Bit 5
            11: outbit <= data_tx[6];          // Bit 6
            12: outbit <= data_tx[7];          // Bit 7
            13: outbit <= 1;              // Stop bit del primer dato
14: outbit <= 0; //inicia transmision del segundo byte
15: outbit <= 1; //dato del segundo byte
            default: outbit <= 1;             // Bad state output idle
        endcase
end

// Output register to pin
assign ser = outbit;

// pruebas de transmision con interruptor

endmodule
```

A.1.11 Convertir hexadecimal a ASCII

```
module Decoder(
input [4:0] data,
output reg [7:0] dec );

always @(*)
begin
  case(data)
    0: dec=48;
    1: dec=49;
    2: dec=50;
    3: dec=51;
    4: dec=52;
    5: dec=53;
    6: dec=54;
    7: dec=55;
    8: dec=56;
    9: dec=57;
    10: dec=97;
    11: dec=98;
    12: dec=99;
    13: dec=100;
    14: dec=101;
    15: dec=102;
    16: dec=0;
  default:dec=48;
 endcase
end
endmodule
```

A.2 Scripts en Matlab

Son los códigos empleados para simulación de la red neurona en el Toolbox de Matlab, generación de archivos de inicialización de memorias del FPGA y comunicación RS-232.

A.2.1 Script en Matlab para entrenamiento de red neuronal

```
clc
clear all
```

```
disp('SIMULANDO....')
si_ent(1,:)= load ('C:\PASalida_Imag_normalizada.txt');
si_ent1(1,:)=si_ent;
si_ent(2,:)= load ('C:\PASalida_Real_normalizada.txt');
si_ent1(2,:)=si_ent;

si_sal(1,:)= load ('C:\PAEntrada_Imag.txt');
si_sal1(1,:)=si_sal;
si_sal(2,:)= load ('C:\PAEntrada_Real.txt');
si_sal1(2,:)=si_sal;

u = con2seq(si_ent1);
y = con2seq(si_sal1);

%definiendo los parametros de la Red
du = [1:4];
dy = [1:4];
narx_pd = narxnet(du,dy,3);
narx_pd.trainParam.epochs = 500;
narx_pd.trainParam.mu = 1;
narx_pd.trainParam.mu_dec = 0.8;
narx_pd.trainParam.mu_inc = 1.5;
narx_pd.layers{1}.initFcn = 'initnw';
narx_pd.layers{2}.initFcn = 'initnw';
narx_pd.divideParam.trainRatio = 40/100;
narx_pd.divideParam.valRatio = 40/100;
narx_pd.divideParam.testRatio = 20/100;

%preparando, entrenando y simulando la red
[p, Pi, Ai, t] = preparets(narx_pd,u,{},y);
narx_pd = train(narx_pd,p,t,Pi,Ai);
y_narx_pd = narx_pd(p,Pi,Ai);
ynarx = cell2mat(y_narx_pd); %salidaNARXentrenada
ynarx(:,1:16)

save('C:\narx_pd.mat','narx_pd')
save('C:\sal_narx.mat','ynarx')


%guardando los pesos y bias
pesosi = narx_pd.IW;
pesosl = narx_pd.LW;
bias = narx_pd.b;

save ('C:\pesos_i.mat','pesosi');
save ('C:\pesos_l.mat','pesosl');
save ('C:\bias.mat','bias');

disp('Finalizado....')
```

A.2.2 Scrip en Matlab para generar archivos "*.coe" de inicialización de memorias del FPGA.

```
clear all
clc
disp('Generando...')
uc(1,:)= load ('C:\PASalida_Imag_normalizada.txt');
ent1(1,:)=uc(1,1:61440);
uc(2,:)= load ('C:\PASalida_Real_normalizada.txt');
ent1(2,:)=uc(2,1:61440);

yc(1,:)= load ('C:\PAEntrada_Imag.txt');
sal1(1,:)=yc(1,1:61440);
yc(2,:)= load ('C:\PAEntrada_Real.txt');
sal1(2,:)=yc(2,1:61440);
```

```
load ('C:\pesos_i.mat','pesosi');
load ('C:\pesos_1.mat','pesosi');
load ('C:\bias.mat','bias');

win1=pesosi{1,1};
win2=pesosi{1,2};
wout=pesosl{2,1};
b1=bias{1,1};
b2=bias{2,1};

tam=size(win1);

for i=1:tam(1,1)
for j=1:2*tam(1,2)
if(j<9)
Pesos_mat(j,i)=win1(i,j);
else
Pesos_mat(j,i)=win2(i,j-8);
end
end
end



[Si_enti,PSei] = mapminmax(ent1(1,5:end));
Si_entri=mapminmax('apply',ent1(1,(1:4)),PSei);
Si_ent_i=[Si_entri Si_enti];


[Si_entr,PSer] = mapminmax(ent1(2,5:end));
Si_entrr=mapminmax('apply',ent1(2,(1:4)),PSer);
Si_ent_r=[Si_entrr Si_entr];


[Si_sali,PSsi] = mapminmax(sal1(1,5:end));
Si_salri=mapminmax('apply',sal1(1,(1:4)),PSsi);
Si_sal_i=[Si_salri Si_sali];


 [Si_salr,PSsr] = mapminmax(sal1(2,5:end));
 Si_salrr=mapminmax('apply',sal1(2,(1:4)),PSsr);
 Si_sal_r=[Si_salrr Si_salr];


In=[Si_ent_i Si_ent_r Si_sal_i Si_sal_r];

formato=16;
%%%%%%%%%% Entradas normalizadas %%%%%%%%%%%%%
fid=fopen('MEM_In.coe','w');
fprintf(fid,'memory_initialization_radix=%i;\n',formato);
fprintf(fid,'memory_initialization_vector= \n');

po=size(In);
for i=1: po(1,2)-1
a=single(In(i));
f=num2hex(a);
fprintf(fid,'%s,\n',f);
end
a=single(In(po(1,2)));
f=num2hex(a);
fprintf(fid,'%s;\n',f);
fclose(fid);

%%%%%%%% Pesos Capa 1%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
for W=1 : tam(1,1)
switch W
    case 1
        fid=fopen('MEM_W1.coe','w');
    case 2
        fid=fopen('MEM_W2.coe','w');
    case 3
```

```
    fid=fopen('MEM_W3.coe','w');
case 4
    fid=fopen('MEM_W4.coe','w');
case 5
    fid=fopen('MEM_W5.coe','w');
case 6
    fid=fopen('MEM_W6.coe','w');
case 7
    fid=fopen('MEM_W7.coe','w');
case 8
    fid=fopen('MEM_W8.coe','w');
case 9
    fid=fopen('MEM_W9.coe','w');
    case 10
        fid=fopen('MEM_W10.coe','w');
    otherwise
end

fprintf(fid,'memory_initialization_radix=%i;\n',formato);
fprintf(fid,'memory_initialization_vector= \n');

for i=1: 2*tam(1,2)-1
a=single(Pesos_mat(i,W));
f=num2hex(a);
fprintf(fid,'%s,\n',f);
end
a=single(Pesos_mat(2*(tam(1,2)),W));
f=num2hex(a);
fprintf(fid,'%s;\n',f);
fclose(fid);
end

%%%%%%%% Pesos Capa 2 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
for W=1 : 2
switch W
    case 1
        fid=fopen('MEM_W21.coe','w');
    case 2
        fid=fopen('MEM_W22.coe','w');
    otherwise
end

fprintf(fid,'memory_initialization_radix=%i;\n',formato);
fprintf(fid,'memory_initialization_vector= \n');

for i=1: tam(1,1)-1
a=single(wout(W,i));
f=num2hex(a);
fprintf(fid,'%s,\n',f);
end
a=single(wout(W,tam(1,1)));
f=num2hex(a);
fprintf(fid,'%s;\n',f);
fclose(fid);
end

%%%%%%% bias %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
fid=fopen('MEM_bias1.coe','w');
fprintf(fid,'memory_initialization_radix=%i;\n',formato);
fprintf(fid,'memory_initialization_vector= \n');

for i=1: tam(1,1)-1
a=single(b1(i));
f=num2hex(a);
fprintf(fid,'%s,\n',f);
end
a=single(b1(tam(1,1)));
f=num2hex(a);
fprintf(fid,'%s;\n',f);
fclose(fid);
```

```
fid=fopen('MEM_bias2.coe','w');
fprintf(fid,'memory_initialization_radix=%i;\n',formato)
fprintf(fid,'memory_initialization_vector= \n');

a=single(b2(1));
f=num2hex(a);
fprintf(fid,'%s,\n',f);
a=single(b2(2));
f=num2hex(a);
fprintf(fid,'%s;\n',f);
fclose(fid);

disp('DONE')
```

A.2.3 Comunicación serial

```
clc
clear
s = serial('COM3','BaudRate',256000);
set(s,'Terminator','NUL');
fopen(s);
for i=1:61436
    i
    g=(fgets(s));
    Data_Imag(i)=hex2num(g);
    g=(fgets(s));
    Data_Real(i)=hex2num(g);
end

im(1:4)=Data_Imag(1);
Re(1:4)=Data_Real(1);
Data_Imag=[im Data_Imag];
Data_Real=[Re Data_Real];

fclose(s)
delete(s)
```

Referencias

[1] D. Morgan, M. Zhengxiang, K. Jaehyeong, M. Zierdt y J. Pastalan, «A Generalized Memory Polynomial Model for Digital Predistortion of RF Power Amplifiers,» *Signal Processing, IEEE Transactions on,* vol. 54, nº 10, pp. 3852-3860, 2006.

[2] M. Rawat, K. Rawat y F. M. Ghannouchi, «Adaptative Digital Predistortion of Wire-less Power Amplifiers/Transmitters using Dynamic Real-Valued Focused Time-Delay Line Neural Network,» *Microwave Theory and Techniques, IEEE Transactions on,* vol. 58, nº 1, pp. 95-104, 2010.

[3] L. Aguilar-Lobo, J. Loo-Yau y S. Ortega-Cisneros, «A Digital Predistortion Technique Based on a NARX Network to Linearize GaN Class F Power Amplifiers,» *IEEE 57th international Midwest Symposium on Circuits and Systems,* 2014.

[4] M. Helaoui y F. Ghannouchi, «Linearization of power amplifiers using the reverse mm-LINC technique,» *Linearization of power amplifiers using the reverse mm-LINC technique,* vol. 57, nº 1, pp. 6-10, 2010.

[5] D. K. Su y W. J. McFarland, «An IC for linearizing RF power amplifiers using envelope elimination and restoration,» *Solid-State Circuits, IEEE Journal of,* vol. 33, nº 12, pp. 2252-2258, 1998.

[6] B. Shi y L. Sundstrom, «Linearization of RF power amplifiers using power feedback,» *Vehicular Technology Conference, 1999 IEEE 49th,* vol. 2, pp. 1520-1524, 1999.

[7] S. C. Cripps, Advanced techniques in RF power amplifier design, Artech House, 2002.

[8] M. Honarvar, M. Moghaddasi y A. Eskandari, «Power amplifier linearization using feedforward technique for wide band communication system,» de *Radio-Frequency Integration Technology, 2009. RFIT 2009. IEEE International Symposium on*, 2009, pp. 72-75.

[9] M. T. Abuelmaatti, A. M. Abuelmaatti, T. Yeung y G. Parkinson, «Linearization of GaN power amplifier using feedforward and predistortion techniques,» de

Quality Electronic Design (ASQED), 2011 3rd Asia Symposium on, 2011, pp. 282-287.

[10] A. N. D'Andrea, V. Lottici y R. Reggiannini, «RF power amplifier linearization through amplitude and phase predistortion,» *Communications, IEEE Transactions on*, vol. 44, nº 11, pp. 1477-1484, 1996.

[11] E. Westesson y L. Sundstrom, «A complex polynomial predistorter chip in CMOS for baseband or IF linearization of RF power amplifiers,» de *Circuits and Systems, 1999. ISCAS'99. Proceedings of the 1999 IEEE International Symposium on*, vol. 1, 1999, pp. 206-209.

[12] D. R. Morgan , Z. Ma, K. Jaehyeong, M. G. Zierdt y J. Pastalan, «A generalized memory polynomial model for digital predistortion of RF power amplifiers,» *Signal Processing, IEEE Transactions on*, vol. 54, nº 10, pp. 3852-3860, 2006.

[13] E. Diaconescu, «The use of NARX neural networks to predict chaotic time series,» *WSEAS Transactions on Computer Research*, vol. 3, nº 3, pp. 182-191, 2008.

[14] D. Pantic, S. Milenkovic, T. Trajkovic, V. Litovski y N. Stojadinovic, «Inverse modeling of semiconductor manufacturing processes by neural networks,» de *Microelectronics, 1995. Proceedings., 1995 20th International Conference on*, vol. 1, 1995, pp. 321-326.

[15] B. O'Brien, J. Dooley y T. J. Brazil, «RF power amplifier behavioral modeling using a globally recurrent neural network,» de *Microwave Symposium Digest, 2006. IEEE MTT-S International*, 2006, pp. 1089-1092.

[16] H. Xie, H. Tang y Y.-H. Liao, «Time series prediction based on NARX neural networks: An advanced approach,» de *Machine Learning and Cybernetics, 2009 International Conference on*, 2009, pp. 1275-1279.

[17] L. Bai y D. Coca, «Nonlinear predictive control based on NARMAX models,» de *In optimization of electrical and electronic equipment* , 2008.

[18] R. Sandrin Ntoune Ntoune, M. Bahoura y C.-W. Park, «FPGA-implementation of pipelined neural network for power amplifier modeling,» de *New Circuits and Systems Conference (NEWCAS), 2012 IEEE 10th International*, 2012, pp. 109-112.

[19] M. T. Hagan y M. B. Menhaj, «Training feedforward networks with the Marquardt algorithm,» *Neural Networks, IEEE Transactions on*, vol. 5, nº 6, pp. 989-993, 1994.

[20] J. Bastos , H. Figueroa y A. Monti, «FPGA implementation of neural network-based controllers for power electronics applications,» de *Applied Power Electronics Conference and Exposition, 2006. APEC'06. Twenty-First Annual IEEE*, 2006.

[21] R. Ntouné, M. Bahoura y P. Chan-Wang, «Power Amplifier Behavioral Modeling by Neural Networks and Their Implementation on FPGA,» de *Vehicular Technology Conference (VTC Fall), 2012 IEEE*, 2012, pp. 1-5.

[22] M. Ownby y W. Mahmoud, «A design methodology for implementing DSP with Xilinx System Generator for Matlab,» de *SOUTHEASTERN SYMPOSIUM ON SYSTEM THEORY*, vol. 35, 2003, pp. 404-408.

[23] T. Possignolo, «Optimized joint NARX ANN-embedded processor design methodology,» de *2009 16th IEEE International Conference on Electronics, Circuits and Systems-(ICECS 2009)*, 2009, pp. 499-502.

[24] «IEEE Standard for Floating-Point Arithmetic (IEEE Std 754-2008),» de *IEEE. Std-754-4985*, 2008.

[25] K. Mohamad, M. F. O. Mahmud, F. H. Adnan y W. F. H. Abdullah, «Design of single neuron on FPGA,» de *Humanities, Science and Engineering Research (SHUSER), 2012 IEEE Symposium on*, 2012, pp. 133-136.

[26] C. Minchola y G. Sutter «A FPGA IEEE-754-2008 Decimal64 Floating-Point Multiplier,» de *Reconfigurable Computing and FPGAs, 2009. ReConFig'09. International Conference on*, 2009, pp. 59-64.

[27] M. Qian, «Application of CORDIC algorithm to neural networks VLSI design,» de *Computational Engineering in Systems Applications, IMACS Multiconference on*, 2006, pp. 504-508.

[28] K. Leboeuf, R. Muscedere y M. Ahmadi, «Performance analysis of table-based approximations of the hyperbolic tangent activation function,» de *Circuits and Systems (MWSCAS), 2011 IEEE 54th International Midwest Symposium on*, 2011, pp. 1-4.

[29] P. Ferreira, P. Ribeiro, A. Antunes y F. M. Dias, «A high bit resolution FPGA implementation of a FNN with a new algorithm for the activation function,» *Neurocomputing*, vol. 71, nº 1, pp. 71-77, 2007.

[30] M. A. Sartin y A. C. Da Silva, «Approximation of hyperbolic tangent activation function using hybrid methods,» de *Reconfigurable and Communication-Centric Systems-on-Chip (ReCoSoC), 2013 8th International Workshop on*, 2013, pp. 1-6.

[31] A. H. Namin, K. Leboeuf, H. Wu y M. Ahmadi, «Artificiàl neural networks activation function HDL coder,» de *Electro/Information Technology, 2009. eit'09. IEEE International Conference on*, 2009, pp. 389-392.

[32] D. Nguyen y B. Widrow, «Improving the learning speed of 2-layer neural networks by choosing initial values of the adaptive weights,» de *Neural Networks, 1990., 1990 IJCNN International Joint Conference on*, 1990, pp. 21-26.

[33] http://www.silabs.com/products/interface/usbtouart/Pages/usb-to-uart-bridge.aspx.

[34] http://www.xilinx.com/support/documentation/data_sheets/ds150.pdf.

[35] [En línea]. Available: http://www.xilinx.com/tools/coregen.htm.

[36] www.xilinx.com/support/documentation/boards_and_kits/ug534.pdf.

[37] www.xilinx.com/support/download/index.html/content/xilinx/en/downloadNav/design-tools.html.

[38] B. Fehri y S. Boumaiza, «Baseband equivalent Volterra series for digital predistortion of dual-band power amplifiers,» *Microwave Theory and Techniques, IEEE Transactions on*, vol. 62, nº 3, pp. 700-714, 2014.

CENTRO DE INVESTIGACIÓN Y DE ESTUDIOS AVANZADOS DEL I.P.N.
UNIDAD GUADALAJARA

El Jurado designado por la Unidad Guadalajara del Centro de Investigación y de Estudios Avanzados del Instituto Politécnico Nacional aprobó la tesis

Implementación Física de la Red Neuronal NARX para Aplicaciones en Predistorsión Digital

del (la) C.

Juan Antonio RENTERIA CEDANO

el día 08 de Agosto de 2014.

Dr. José Raúl Loo Yau
Investigador CINVESTAV 3B
CINVESTAV Unidad Guadalajara

Dra. Susana Ortega Cisneros
Investigador CINVESTAV 3A
CINVESTAV Unidad Guadalajara

Dr. Luis Ilich Vladimir Guerrero Linares
Docente Investigador
Centro de Enseñanza Técnica Indusrial